NASA Contractor Report 4509

Using Penelope To Assess the Correctness of NASA Ada Software: A Demonstration of Formal Methods as a Counterpart to Testing

Carl T. Eichenlaub, C. Douglas Harper,
and Geoffrey Hird

CONTRACT NAS1-18972
MAY 1993

NASA

NASA Contractor Report 4509

Using Penelope To Assess the Correctness of NASA Ada Software: A Demonstration of Formal Methods as a Counterpart to Testing

Carl T. Eichenlaub, C. Douglas Harper,
and Geoffrey Hird
ORA Corporation
Ithaca, New York

Prepared for
Langley Research Center
under Contract NAS1-18972

National Aeronautics and
Space Administration

Office of Management

Scientific and Technical
Information Program

1993

Abstract

Life-critical applications warrant a higher level of software reliability than has yet been achieved. Since it is not certain that traditional methods alone can provide the required ultra-reliability, new methods should be examined as supplements or replacements. This paper describes a mathematical counterpart to the traditional process of empirical testing.

ORA's Penelope verification system is demonstrated as a tool for evaluating the correctness of Ada software. Grady Booch's Ada calendar utility package, obtained through NASA, was specified in the Larch/Ada language. Formal verification in the Penelope environment established that many of the package's subprograms met their specifications. In other subprograms, failed attempts at verification revealed several errors that had escaped detection by testing.

Keywords

Life-critical software, ultra-reliability, formal methods, formal specification, formal verification, safety

Contents

Preface

This report provides computer science professionals with a demonstration of Penelope, ORA's program verification system for producing highly reliable Ada components. The reader familiar with the Ada language will find the document self-explanatory. No prior knowledge is required of program verification in general or Penelope in particular: the basic concepts of program verification are explained before they are applied.

From this document, the reader should obtain a basic understanding of the capabilities and methods of the Penelope system when used as a counterpart to testing. The reader should also see that evaluating completed software is only one of many possible applications of program verification. The reader who wishes to explore the possibilities is directed to the suggestions for further reading.

Summary

Ultra-reliable software for life-critical aerospace applications is NASA's goal. Specifically, the FAA proposes a probability of 10^{-9} of catastrophic failure during one service hour. Conventional software development methods have not by themselves produced ultra-reliable software. Several fly-by-wire systems now in service have failed catastrophically. Improvements to conventional methods need to be investigated. Promising alternative approaches deserve examination as well. Formal verification applies the power and precision of mathematics to the development of life-critical software, in such a way that software can be unambiguously specified and conclusively proved to meet its specifications. ORA has developed Penelope, a computerized system for formally verifying Ada software. This report demonstrates Penelope's capabilities.

NASA provided ORA with Grady Booch's `Calendar_Utilities`, a commercially released utility package, suitable for navigational uses. The package features several convenient representations of time and date and manipulations of them. ORA formally specified `Calendar_Utilities` using the Larch/Ada mathematical specification language. ORA formally verified `Calendar_Utilities` using the Penelope verification environment.

Formal specification provided an unambiguous description of every subprogram in the package. Informal analysis then uncovered dubious software and three outright errors. Formal verification determined the status of eleven representative subprograms. Successful proofs demonstrated nine subprograms to be correct. In two subprograms, failed attempts at verifications revealed additional errors in human blind spots.

Penelope is an effective, prototype tool which holds promise for producing ultra-reliable Ada software. As ORA continues development, Penelope will become even more effective.

1 Introduction

As the complexity of life-critical computer applications increases and the reliance placed upon these applications grows, software developers will have to attain higher levels of reliability than have been possible to date. Since it cannot be foreseen whether today's software practices will scale up to meet the challenge, alternative approaches need to be examined. One promising approach is the use of formal verification to prove mathematically that software is correct. This paper demonstrates that the formal approach is practical for the evaluation of existing software.

1.1 The Challenge: Ultra-Reliability

Nowhere are computerized life-critical systems more pervasive than in aerospace. Today, there are aircraft that cannot be controlled without computer assistance. In the future, there will be vehicles flown completely by computer. It is imperative that their software be correct.

The Federal Aviation Authority [4] has proposed a very ambitious standard for flight safety: a probability of 10^{-9} of catastrophic failure during a one-hour flight. Meeting this standard would effectively eliminate risk from flight. The probability of a fatal failure during 1000 years' worth of consecutive independent one-hour missions would be less than one percent.

Can the FAA's goal be achieved? The proposed ultra-reliability is several orders of magnitude from what has ever been attained for aircraft with purely human control. Computerized assistance, at the least, will be required if the FAA's goal is to be met. Ultra-reliable software will be needed for life-critical applications, but it is not at all clear that conventional software development methods alone are capable of its production.

1.2 The Conventional Approach

For a system to behave reliably, its developers must know what constitutes correct behavior: a specification[1] is necessary. Traditional English-language specifications of the intended behavior of the software suffer from the imprecision and ambiguity inherent in natural language. Consequently, misunderstandings and resultant errors are common. Ultra-reliability has never been attained for software specified by traditional means.

But suppose that developers could attain ultra-reliability by some modification of traditional methods. How would they convince an impartial skeptic that they had done so? The faith of the developers in their product is not sufficient. Certification is not a matter of faith, but of demonstration.

[1] By *specification* in this report, we mean a description of what the software does. If we intend the Ada Reference Manual's sense of specification, we use the phrase *Ada specification*.

Conventional demonstrations of reliability depend heavily on empirical testing, a costly and time-consuming process that is never fully completed. Testing can conclusively demonstrate the existence of a particular error, but cannot demonstrate the general absence of all errors. Prolonged testing of software of any real complexity will continually reveal flaws for as long as tests are carried out. Error-free software is approached, at best, only asymptotically by testing.[2]

The conclusions from testing about reliability are therefore drawn in terms of probability. Techniques are available that enable extrapolation from relatively short tests to probabilistic predictions of relatively long intervals of failure-free operation; however, these techniques do not scale up gracefully to encompass intervals of thousands of years. An alternative to empirical testing is the formal approach.

1.3 The Formal Approach

By applying mathematics to the specification and demonstration of correctness of software, great advances in reliability are possible. The use of a formal specification language brings the clarity and precision of mathematical logic to the description of the software's intended behavior. Formal specifications provide the user and the implementer with an unambiguous means of communication.

Formal verification is a mathematical counterpart to empirical testing. The verifier proves mathematically that software meets its specifications in all cases, rather than testing particular cases. Where testing is specific, verification is general. The tester removes flaws from software one by one. The verifier develops software free from flaws.

A leader in formal methods, the ORA Corporation has developed the Penelope [8, 14, 16] verification system for specifying and verifying Ada software. Using Penelope, computer science professionals can ensure that software is of the highest reliability. This paper demonstrates how.

1.4 Demonstrating the Formal Approach with Penelope

NASA Goddard Space Flight Center and NASA Lewis Research Center furnished ORA with samples of Ada software. From these samples, we chose a demonstration case: Grady Booch's `Calendar_Utilities`,[3] a commercially available software package. We formally specified the subprograms of the package in the Larch/Ada language, then formally evaluated the correctness of the software using the Penelope editor. We verified that many of the subprograms are correct, but also uncovered errors in the software during both the specification and verification efforts.

[2] In practice, in correcting one error, the developer risks introducing others.

[3] Copyright Grady Booch, part of the Ada Booch components 1984-1992.

1.4.1 The Evaluated Software

`Calendar_Utilities` is an extension of the basic `Calendar` package built into Ada, and employs `Calendar`, along with several other packages, by means of `with` clauses. The data types of `Calendar_Utilities` provide several natural representations for time and date, including string representations. The subprograms of the package convert from one representation of time to another, extract specific information (such as day of the week) from a time value, and perform arithmetic on time values. `Calendar_Utilities` encapsulates a rich set of manipulations of dates between 1901 and 2099.

1.4.2 The Specification

We used the Larch/Ada specification language to state fully and precisely the intended behavior of each of the subprograms in the package. On the basis of the specifications, we informally examined and evaluated the software. We detected outright errors, dubious features, and hidden assumptions.

1.4.3 The Verification

In the second phase, we selected subprograms from the package to evaluate within the Penelope Ada verification environment. We proved several subprograms to be correct; we showed others to be incorrect. We confirmed some errors we had detected earlier, during the specification phase, and detected other errors. The nature of the newly found errors was telling: they had evaded earlier detection. Penelope uncovered these errors because the formal approach is rigorous and mathematical.

1.5 Structure of the Document

Section 2 is a self-contained overview of program verification in the Penelope system; the overview is provided for readers with no prior exposure to formal methods. Before discussing the demonstration of Penelope in detail, we describe the subject of the demonstration. Section 3 explains the software that was evaluated, the package `Calendar_Utilities`. In Section 4 we demonstrate formal specification and verification in Penelope. We draw our conclusions in Section 5.

Appendix A describes related work. Appendix B is the glossary for this paper. Appendix C contains the original text of the package `Calendar_Utilities`. For convenient reference, Appendix D contains the text of the built-in Ada language `Calendar` package as it appears in the Ada Reference Manual. Finally, Appendix E contains the full Ada and Larch/Ada text of the software, specification, and verification performed with Penelope.

2 Overview of the Penelope System

2.1 Basics of Formal Specification and Verification

Every software developer reasons about program semantics. When writing software, the developer considers, usually on a very informal level, what constitutes correct behavior and how to make the software behave correctly. Sometimes, however, developers reason semi-formally about the behavior of software through thought experiments in which programs are "executed" with paper and pencil.

Consider the familiar three-step swap of `x` and `y`:

```
t := x;
x := y;
y := t;
```

The following symbolic execution shows that the software does indeed interchange the values of the variables `x` and `y`. The intermediate values between execution steps reveal the action of the software.

	$x = x_{init}$	$y = y_{init}$	
`t := x;`			
	$x = x_{init}$	$y = y_{init}$	$t = x_{init}$
`x := y;`			
	$x = y_{init}$	$y = y_{init}$	$t = x_{init}$
`y := t;`			
	$x = y_{init}$	$y = x_{init}$	$t = x_{init}$

A formal verifier reasons much the same way, but does so at the level of mathematical rigor, working within a mathematical model of computation for the computer language.

In the first step of formal specification and verification, the verifier uses the mathematical language of the model to describe the intended semantics of the software (what the software should do). This description is the formal specification. Next, the verifier analyzes the software within the model to derive the actual semantics (what the software actually does); this analysis is the first step of the formal verification. In the final step of the verification, the verifier uses formal mathematical proofs to show that the actual semantics are the intended semantics; that is, that the software does what the software developer intended. We elaborate on each of these steps in the following subsections.

2.1.1 Formal Specification

Formal specifications of software are written in a formal language having a precisely defined syntax and semantics. The principal benefits of formal speci-

fication are the following:

- Formal specifications explicitly state all underlying assumptions.
- Formal specifications can be stated at a high level of abstraction.
- Formal specifications provide unambiguous descriptions.
- Formal proofs of correctness are possible.
- Formal specification and the subsequent formal verification allow enhanced confidence for critical systems.

The fundamental notion of program semantics is that of change of state; that is, the alteration of data. To specify a program formally is to describe mathematically how it transforms one state into another. We describe the states of a transformation by means of *assertions*, mathematical statements about those states.

We assume that a program is called in an initial state, performs a set of instructions, and terminates.[4] Executing the program transforms an initial system state into a final state: program variables that hold certain values in the initial state may acquire new values in the final state, information in external storage is changed through I/O, and so on. We could completely describe the semantics of terminating programs by giving an exhaustive list of all possible pairs of initial system states and the final states into which the software transforms them.

An exhaustive description of program semantics would be similar to the complete description of a mathematical function by the list of all ordered pairs $(x, f(x))$ of its arguments and values. The complete listing is fully adequate, but a formula for $f(x)$ is vastly preferable. For most purposes, the advantages of the formula $f(x) = x^2$ over the list (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), ... are obvious.

Likewise, a symbolic way to relate possible initial states to possible final states is vastly preferable to a mere listing of pairs of states. The symbolism most often used in the theory of formal verification to relate initial states, software, and final states is the Floyd-Hoare triple [12], a kind of assertion. The triple

$$\{precondition\}\, S\, \{postcondition\}$$

symbolically states that the software S transforms the system from an initial state in which the assertion *precondition* holds to one in which the assertion *postcondition* holds.

In this notation, the swap example above is described by:

$$\{\mathtt{x} = x_{init} \text{ AND } \mathtt{y} = y_{init}\}\, S\, \{\mathtt{x} = y_{init} \text{ AND } \mathtt{y} = x_{init}\}$$

[4] We are specifically excluding programs that are intended to run indefinitely, such as operating systems, I/O repeaters, and so on.

The triple asserts that S, the software that performs the swap, transforms any given initial state in which $\mathbf{x} = x_{init}$ and $\mathbf{y} = y_{init}$ into a final state in which $\mathbf{x} = y_{init}$ and $\mathbf{y} = x_{init}$. The triple above is a specification for the swap S.

Floyd-Hoare triples have the same advantages for specifying program semantics that formulas have for defining functions. Instead of listing all possible behaviors separately, we can describe the entire collection of permissible behaviors symbolically.

2.1.2 Formal Verification

Having made a specification of the intended semantics in terms of pre- and postconditions,

$$\{precondition\}\ S\ \{postcondition\}$$

the verifier mathematically analyzes the software for correctness. Analyzing for correctness is the process of formally determining whether S is, in fact, a state changer that brings about *postcondition* whenever *precondition* is initially true. The two parts to this process are predicate transformation and proofs.

Predicate Transformation The first part of formal verification is a kind of symbolic execution known as *predicate transformation.*[5] Similar to paper-and-pencil execution in reverse, predicate transformation works from *postcondition*, the stated goal of the software, back to the essence of what must be true before S is executed. Predicate transformation calculates the *weakest precondition*, i.e., the most general (hence, weakest in the logical sense) precondition that is certain to be transformed by S into *postcondition*. [6] The notation for the weakest precondition is

$$wp(S, postcondition)$$

The weakest precondition is the precondition in the Floyd-Hoare triple that describes the actual semantics of S:

$$\{wp(S, postcondition)\}\ S\ \{postcondition\}$$

Any state satisfying $wp(S, postcondition)$ will be transformed by the action of S into a state satisfying *postcondition*. By calculating the weakest precondition, the verifier derives the actual semantics of the software within the mathematical model.

Let us return to the example of the three-step swap S as specified above. Reading from the bottom up, here are the intermediate and ultimate results of predicate transformation of *postcondition* through S. As a careful reading

[5] In mathematical parlance, the precondition and postcondition are *predicates* on the states.

[6] The expert in formal methods will note that, as Larch/Ada specifications are *partial correctness* specifications, we are actually interested in the "weakest *liberal* precondition"; and, in addition, our predicate transformers compute an approximation to the weakest liberal precondition. For the purposes of this discussion, these distinctions can be ignored.

shows, at each intermediate step, predicate transformation has calculated what must be true if the remaining code is to establish *postcondition*.

	{y = y_{init} AND x = x_{init}}	*wp*(*S*, *postcondition*)
t := x;		
	{y = y_{init} AND t = x_{init}}	(2)
x := y;		
	{x = y_{init} AND t = x_{init}}	(1)
y := t;		
	{x = y_{init} AND y = x_{init}}	*postcondition*

Formal Proof The second part of formal verification is the demonstration that the calculated actual semantics agree with the specified intended semantics.

When the specified precondition logically implies the calculated weakest precondition, the semantics do agree. The specified precondition will be true in every initial state allowed by the specification. From the logical implication, it follows that the weakest precondition will also hold in every initial state allowed by the specification. From the definition of weakest precondition, it follows that S brings about the postcondition in the final state, as desired. Thus the actual behavior is the intended behavior.

The formula relating the specified and weakest preconditions is the *verification condition* or *VC*:

$$precondition \rightarrow wp(S, postcondition)$$

Proving the verification condition verifies the software.

To illustrate, we show the VC of the three-step swap above:

$$(x = x_{init} \text{ AND } y = y_{init}) \rightarrow (y = y_{init} \text{ AND } x = x_{init})$$

Since the conclusion of the VC differs from the hypothesis only by rearrangement, the VC is easy to prove. The software is formally verifiable.

2.1.3 Summary

- The specified precondition and postcondition describe the intended behavior of the software.
- Predicate transformation is symbolic execution for the purpose of determining the actual semantics of the software.
- The logical formula connecting the actual semantics and the intended semantics is the verification condition (VC).
- Proving the VC shows that the software's actual semantics are the intended semantics.

2.2 Formal Specification in Larch/Ada

We have described, in general, formal specification using Floyd-Hoare triples. This section outlines formal specification using the Larch/Ada language. Larch/Ada consists partly of an adaptation of the Floyd-Hoare notation, and partly of a means for defining mathematical vocabulary. Its written form is designed to be readable by the Penelope verification environment. In this section, we show a Larch/Ada specification as it looks in Penelope.

2.2.1 Mathematics and Programming

There are both mathematical and programming expressions in a Floyd-Hoare triple. In a triple

$$\{\textit{precondition}\}\ S\ \{\textit{postcondition}\}$$

the statement S is written in a programming language (in this case Ada), while the precondition and postcondition are written in the specification language (in this case Larch/Ada).

Although some programming languages are mathematically oriented, they are not purely mathematical. The essential difference is that programming languages are executable while mathematical language is descriptive.

Consider for example the Ada operation `+` and the mathematical operation $+$. The ada expression `x + y` invokes a computation that may return a value, but may also raise a numeric exception if the arguments are too large. The mathematical expression $x + y$, on the other hand, is completely descriptive: it simply denotes the sum of the values of the variables. The Ada expression is an instruction; the mathematical expression is a description.

In Larch/Ada, as in all specification languages in the Larch [10] family, the distinction between the two languages is carefully maintained. Larch/Ada cleanly separates mathematical and programming concerns. Mathematics is done in a language in which terms have their ordinary mathematical meanings. Programming is done in Ada, where expressions have their ordinary computational effects.

2.2.2 The Two-Tiered Approach

Larch/Ada, following the the Larch paradigm, has two major divisions or *tiers*: one for pure mathematics, one for an interface between mathematics and programming. The mathematical tier defines the mathematical vocabulary used in the specification. The interface tier specifies the behavior of programs.

The Mathematical Tier In the mathematical tier we employ the Larch *Shared Language*, common to the entire Larch family, to group together declarations of kinds of mathematical objects, declarations of functions, and definitions of objects and functions into collections called *traits*. In a trait, we first

declare *sorts*—collections of abstract mathematical objects. There are sorts of integers, of lists, of stacks, of queues, of trees, and so on. We next declare the functions that operate on the sorts. Examples of functions from the sort of stacks are *push*, *pop*, *top*, and so on. We then define the functions mathematically with axioms and lemmas. An example is the axiom on stacks, $pop(push(element, St)) = element$.

The Interface Tier In the interface tier, we use the Larch/Ada interface language to describe the correspondence between the mathematical and programming realms. Part of the correspondence is built into Larch/Ada. Ada types are *based on* the Larch sorts modeling them. That is, Ada objects of a given type take their values in the corresponding Larch sort. For instance, Ada type `integer` is based on the mathematical integers of Larch sort `Int`. The set of all possible values of the type form a subset of the sort. The possible values of Ada type `integer` range from `minint` to `maxint` within sort `Int`, which is infinite.

Two-Tiered Specification Building on this foundation, we specify the software. There are special idioms for specifying the values of objects, for specifying the conditions that must hold when a program is called, for specifying what will be true upon normal exit, and for specifying the circumstances under which exceptions will be propagated and the consequences of doing so.

An example of the interplay of the two tiers is shown below in the Larch/Ada specification of `fact`, an Ada function to compute factorials. In Penelope, Larch/Ada appears as Ada commentary, set off with "--|". The specification below shows a Larch trait, the Ada function, and the pre- and postconditions of the triple after the function.

```
--| TRAIT T IS
--| INTRODUCES
--| factorial: Int-> Int;
--| AXIOMS: FORALL  [m:Int]
--| f0: factorial(0) = 1;
--| fplus: (m > 0) -> (factorial(m) = m*factorial(m-1));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;

FUNCTION fact(n : IN integer) RETURN integer;
  --| WHERE
  --|       IN (n>=0);
  --|       RETURN factorial(n);
```

```
--| END WHERE;
```

Trait `T`, appearing before the Ada function, introduces the mathematical factorial function, then defines it through axioms `f0` and `fplus`.

Note that because we use the ASCII character set for machine-readable Larch/Ada, our mathematical language looks somewhat like a programming language. There is no possibility for confusion, however, if the reader bears in mind that the "`--|`" marker precedes mathematical notation.

The vocabulary newly defined in trait `T` is used in the Larch/Ada interface, between "`WHERE`" and "`END WHERE`".

The `IN` annotation gives the precondition of `fact`: the value of the argument `n` must be non-negative.

The `RETURN` annotation gives the postcondition: the value returned by the function must be `factorial(n)`.

Note: While Larch/Ada has idioms for side-effects on globals, and for termination by propagating exceptions, those idioms do not appear here. Their absence means that `fact` must have no side effects and must not terminate exceptionally.[7]

2.2.3 Summary

- Larch/Ada cleanly separates mathematical and programming concerns.
- The Larch/Ada mathematical tier provides the mathematical vocabulary for the interface tier.
- The Larch/Ada interface tier makes explicit connections between mathematics and programming, specifying program behavior in mathematical terms.

[7]This statement is almost correct as it stands. Please see the discussion of correctness in Section 4.4.1.

2.3 Formal Verification in Penelope

We have described, in general, formal verification using predicate transformation and proofs of VCs. This section outlines formal verification using the Penelope system. In this section, we show how we use Penelope to evaluate a piece of software.

2.3.1 Computer-Assisted Verification

Penelope is an environment for verifying Ada programs. From Ada input and Larch/Ada specifications, Penelope automatically performs the predicate transformations, calculating the pre- and postconditions and producing verification conditions. The verifier then uses Penelope's built-in proof editor to prove the VCs.

The entire process is interactive and incremental. Any user of a spreadsheet will find Penelope's approach familiar. The verifier edits the software, the specifications, or the proofs. Each change is propagated throughout the file by predicate transformation. Viewing the recalculated results, the verifier makes another change. The process continues until the verifier has proved all the VCs. The final product is verified, legal Ada code.

2.3.2 Larch/Ada in Penelope

As we mentioned in the last section, Larch/Ada appears in a Penelope file as Ada commentary, set off by the special mark "--|". Likewise, displayed preconditions, VCs, and proofs are Ada commentary. Thus the output of Penelope will always be a legal Ada program and verified output will be a correct Ada program.

2.3.3 Evaluating Existing Software

Proving Correctness When we use Penelope to verify an existing Ada program, we must add our Larch/Ada specifications to the Ada program. As we do so, Penelope interactively computes and displays the VCs that we must prove. When we have proved them, we have formally proved that the software meets its specification.

Detecting Errors When we cannot prove program VCs, it may be because the software does not meet its specification or because we simply have not thought of the correct approach to the proof.

However, when the software is incorrect, we can usually find out by examining the proofs. Having located an error this way, we may confirm it with a special-purpose test run of the software. Penelope augments testing, rather than replacing it. Formal verification can demonstrate correctness; testing can demonstrate incorrectness.

Below, we show how we used Penelope to evaluate two versions of the Ada procedure **swap**: one correct, one incorrect. Both versions have the same specification:

```
PROCEDURE swap(x, y : IN OUT integer)
  --| WHERE
  --|       OUT ((x=IN y) AND (y=IN x));
  --| END WHERE;
```

If we had wanted to restrict the initial state, we would have explicitly specified the precondition with an **IN** annotation. Not giving one implicitly specifies a precondition of **TRUE**. Since **TRUE** holds in any state, we are allowing any initial state for **swap**.

With the Larch/Ada **OUT** annotation, we are specifying that the final value of **x** is the initial value of **y**, and vice versa.

It is trivial to verify the correct version. The completed proof of the VC follows the specification, set off by the marker "--!". The computed precondition of the procedure body is also set off this way.

```
--! Verification status: Verified
PROCEDURE good_swap(x, y : IN OUT integer)
  --| WHERE
  --|       OUT ((x=IN y) AND (y=IN x));
  --| END WHERE;
  --! VC Status: proved
  --! BY synthesis of TRUE
  --! []

IS
  t : integer;

BEGIN
  --! PRECONDITION = ((y=IN y) AND (x=IN x));
  t:=x;
  x:=y;
  y:=t;
END swap;
```

After we entered our specification, Penelope calculated the precondition of the procedure body, and from the precondition, Penelope calculated the VC. Since **x** = **IN x** in the initial state (and likewise for **y**) Penelope was able to simplify the VC to **TRUE** before displaying it for proof. We proved the trivial VC immediately. Since the proof is completed, Penelope does not display the VC.

We may locate the error in the incorrect version by inspecting the VC:

```
--! Verification status: Not verified
PROCEDURE bad_swap(x, y : IN OUT integer)
  --| WHERE
  --|       OUT ((x=IN y) AND (y=IN x));
  --| END WHERE;
  --! VC Status: ** not proved **
  --! >> (x=y)
  --! <proof>
  --! []

IS
  t : integer;

BEGIN
  --! PRECONDITION = ((x=IN y) AND (x=IN x));
  t:=x;
  y:=t;
  x:=y;
END swap;
```

Having no assumptions about **x** and **y** in general, we could not prove the VC. Looking at the code, we saw that the equation $\mathbf{x} = \mathbf{y}$ arises from the incorrect order of the assignments. The program will always set **x** and **y** to the same value: namely, **IN x**.

2.3.4 Summary

- The user of Penelope enters Ada software and Larch/Ada specifications.
- Penelope computes preconditions and VCs.
- The user edits proofs of the VCs. Penelope checks the proofs for correctness.
- When all VCs are proved, the Ada software is verified.
- Unprovable VCs are evidence of errors.

3 Test-Case Software

So that we could demonstrate formal specification and verification in Penelope on actual software, rather than artificial examples, NASA furnished ORA with a commercially obtained copy of Grady Booch's `Calendar_Utilities` package. This package provides several convenient ways to represent and manipulate times and dates between the years 1901 and 2099 A.D., going considerably beyond what is built into the Ada language. The full Ada text of `Calendar_Utilities` appears in Appendix C. Here, we describe the package conceptually.

3.1 Data Types

Dates and times are expressed in units of years, months, days, hours, minutes, seconds and milliseconds.

A date can be expressed two ways. The first is by giving the year, the month, and the day of the month. The second is by giving the year and the day within the year. Both (1991, February, 2) and (1991, 33) express the same date.

The expressions of date and time are represented by three kinds of data type: numeric, enumerated, and string.

3.1.1 Numeric Representations

There are two explicit numeric representations of date and time, and one implicit representation of date.

- Type `Time`

  This record type represents a date and time by year, month, day, hour, minute, second, and millisecond.

- Type `Interval`

  This record type represents passage of time in units of days, hours, minutes, seconds, and milliseconds.

- Year and Day (Implicit)

  Several functions of the package have **`the_year`** and **`the_year_day`** in their argument lists to represent the date.

3.1.2 Enumerated Representations

Names of the months and days of the week are represented by enumeration types.

- Type **Month_Name**

  (January, February, March, April, May, June, July, August, September, October, November, December)

- Type **Day_Name**

  (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)

Note that the week is unconventionally listed from Monday instead of its first day, Sunday. This unconventionality can lead to confusion in reading the code of the package unless it is kept in mind.

3.1.3 String Representations

Dates and times represented by strings can each have two formats.

- Date
  - Full (US standard)
    Example: `"March 2, 1991"`
  - Month-Day-Year
    Example: `"03/02/91"`
- Time
  - Full (hours, minutes, seconds, milliseconds, am/pm)
    Example: `"01:23:45:67 pm"`
  - Military (24 hour clock)
    Example: `"13:23:45:67"`

3.2 Operations

There are two categories of operations on data types: extractors of information and converters between representations. We give informal, English descriptions of the operations here. We formally specify them in Section 4.2.2.

When two or more functions share a name, we briefly indicate in a parenthetical comment what kind of argument each function takes.

3.2.1 Extracting Information

- **Is_Leap_Year**

  This Boolean function tells whether a given year is a leap year.

- **Days_In** (Year)

  This function calculates the number of days in a given year.

- `Days_In` (Month)

  This function calculates the number of days in a given month.

- `Day_Of` (Year, Day)

  This function calculates the day of the week of a given year-day pair.

- `Day_Of` (Time)

  This function calculates the day of the week of a date and time given in the representation of type `Time`.

- `Period_Of`

  This function tells whether a given date/time is a.m. or p.m.

3.2.2 Converting Between Representations

- `Month_Of` (Month Number)

  This function converts the representation of months from numeric to enumerated.

- `Month_Of` (Month Name)

  This function converts the representation of months from enumerated to numerical.

- `Time_Of` (Year, Day)

  This function converts dates from the year-day representation to the `Time` representation.

- `Time_Of` (Time)

  This function converts time from the `Calendar_Utilities Time` representation to the built-in Ada `Calendar.Time` representation.

- `Time_Of` (Calendar.Time)

  This function converts time from the built-in Ada `Calendar.Time` representation to the `Time` representation.

- `Date_Image_Of`

  This function converts dates from the `Time` representation to a string representation.

- `Value_Of` (Date/Time Strings)

  This function converts date and time from a string representation to the `Time` representation.

- **`Duration_Of`**

  This function converts time from the **`Interval`** representation to the built-in Ada **`Duration`** representation.

- **`Interval_Of`**

  This function converts time from the built-in **`Duration`** representation to the **`Interval`** representation.

- **`Image_Of`**

  This function converts time from the **`Interval`** representation to a string representation.

- **`Value_Of`** (Interval String)

  This function converts time from a string representation to an **`Interval`** representation.

4 A Demonstration of the Penelope System

This section describes our formal verification of the `Calendar_Utilities` package using Penelope. First, we present an overview of the demonstration. Next, we describe the specifics of the formal specification. Last, we give a complete account of the formal verification.

4.1 Overview of the Demonstration

During the first phase of this project, we developed Larch/Ada specifications for every subprogram in the `Calendar_Utilities` package.[8] These specifications provided the context for informal mathematical discussion of the correctness of the subprograms. We noted in the interim report that most of the subprograms appeared to be correct, but that three subprograms were suspect.

In the second phase of this project, we used the Penelope Ada verification environment to investigate correctness formally. Our intention was to give a full, mathematically sound treatment of as much of the Ada code as was feasible with the resources available. We largely accomplished our intentions, but we must state some reservations. We found that, even on a Sun SPARCstation[9], a verification of the entire package at one time was beyond the current capabilities of Penelope.

The bookkeeping information for each individual verification was kept in the buffer at at all times, seriously slowing the editor. The upcoming version of Penelope solves this problem, supporting libraries of verified components. After we verify a function, we will be able to store its verified text in a library and eliminate the bulk of its text from the buffer. Any time we need information about the function, we will retrieve it from the library.

We were also hampered by Penelope's incomplete support for strings. Penelope supports only unconstrained arrays and does yet not have string literals. We acknowledged both limitations and narrowed our scope to those subprograms in the package that do not involve strings. Six subprograms of twenty involve strings; of the remaining fourteen, we treated eleven in complete detail.

To perform the verification, we needed to expand both the Ada and the Larch/Ada text beyond what we presented in the interim report. These expansions were in three categories: additional specified Ada code required for a complete verification; additional theoretical facts; and additions and modifications to the Ada code necessary to circumvent limitations of Penelope.

The specification required enlargement. There are several dependencies in `Calendar_Utilities` involving many calls to subprograms in other packages. To represent formally the effect of these calls within Penelope, we needed to specify the called subprograms. For each package upon which `Calendar_Utilities`

[8] Copyright Grady Booch, part of the Ada Booch components 1984-1992.

[9] Sun is a trademark of Sun Microsystems, Inc. SPARCstation is a trademark of SPARC International, Inc.

depends, we needed an (Ada) package specification, fully specified in Larch/Ada.

We also had to enlarge the mathematical theory under which we verified `Calendar_Utilities`. Our specification included several facts of arithmetic that are not built into Penelope and had to be explicitly stated in the traits. Among these traits are certain facts about modular arithmetic, rational numbers, etc.

The last expansion included additions and modifications to the Ada code. Most of the modifications to the verified Ada code were slight. One example is the translation of `FOR` loops into `WHILE` loops. Penelope will support `FOR` loops in the next release. Another example involves type conversion. Because Penelope does not currently support type conversion, we wrote special-purpose Ada functions whose annotated effect is to convert objects of one type into objects of another. The major modification was our treatment of the fixed-point type `duration` as a rational number type, constructed as a record. The reasons for this modification are described below in Section 4.4.1.

The formal treatment of `Calendar_Utilities` was the largest undertaking yet for Penelope: 2938 lines[10] of code, annotation, and verification, of which 441 lines were Ada code. Both the specification and verification phases of the project have demonstrated their value in assuring correctness and detecting errors in actual Ada programs.

[10] before reformatting for this report

4.2 Specification

In Section 2, we described the general aspects of formal specification and the Larch/Ada language. In this section, we discuss Larch/Ada traits further, and we present detailed specifications of the `Calendar_Utilities` subprograms.

4.2.1 Mathematical Preliminaries

The specification of a program is written in the context of a logical theory defined by a single main trait, in which all of the mathematical vocabulary is introduced and defined. In a modular fashion, the main trait is built up from smaller component traits. Any trait may **include** another trait as a component, using a facility much like the Ada **with** facility for library unit dependencies. In this section, we describe the various component traits used for the specification.

Component traits comprise two classes: a class of traits that the system implicitly generates in response to the Ada text, and a class of traits that the user explicitly constructs.

System-Generated Traits Just as an Ada program runs within the environment of the package **standard**, the main Larch trait implicitly includes (as though with an **include** statement) a standard trait that defines Ada's built-in data types. The standard trait defines the following: the integer sort `Int` and its associated operations (+, *, etc.); the enumeration sort AdaBool with its literals (`TRUE`, `FALSE`) and its operations (`AND`, `OR`, `NOT`, etc.); and other basic sorts, objects, and operations. Ada type **integer** is based on Larch sort `Int`, **boolean** is based on `AdaBool`, and so on. This standard trait essentially gives the theory governing Ada package **standard**.

In addition to the standard trait, the system implicitly generates other traits to provide the abstract data sorts on which Ada's constructed types are based, together with the theory of those sorts. For example, if a user declares the Ada type

```
TYPE color IS (red, amber, green);
```

the system implicitly generates a Larch/Ada trait and implicitly includes the trait in the main trait for the Ada program. This trait makes available a new enumerated sort on which to base `color`, together with the theory that makes it the proper enumerated sort.

In general, when a type `T` occurs in a program, Penelope creates the sort on which `T` is based. The mathematical operations on the sort use similar syntax to Ada operations on `T`, and so can be understood with little difficulty.

The Penelope system generates an internal name for each sort it creates. For example, the internal name for the sort on which the enumeration type `color` is based, is essentially a listing of all its enumeration literals

```
(red, amber, green)
```

Since such internal names can become unwieldy, we have added notational extensions to Larch by which the user may introduce more convenient names for these sorts. A full description of implicitly generated traits and our notational extensions to Larch is given in ORA Technical Report TR 89-34 [7].

User-Constructed Traits The user can construct traits to gain additional expressiveness or convenience. For example, in specifying `Calendar_Utilities`, we constructed traits to introduce new sorts, to give convenient names to existing sorts, to define conversion factors between time units, and to define various auxiliary functions from one time representation to another.

Traits contain both axioms and lemmas. The axioms express the theory in the form best suited to mathematical exposition. The lemmas encapsulate essential facts of the theory in a form best suited to proving VCs. In constructing our traits, we tried to write the axioms and lemmas in a style that appeals directly to people's usual understanding of the calendar. For example, when specifying the predicate `is_leap_year`, our aim was to provide a more straightforward notion of a leap year than the code for that function can provide and a more clearly stated specification than dense mathematics would provide.

Current Limitations of Traits in Penelope While the mathematical theory underlying Penelope is mature and adequate for formal verification, Penelope itself is still being developed. Consequently, we had two limitations on traits as input.

The first limitation is Penelope's syntax for traits. The Penelope syntax for traits is an extract of and somewhat different from the syntax in the Larch manual, although most of the differences are minor, reflecting the embedding of traits as commentary in Ada code.

The other limitation involves sortchecking of the logical language. Penelope's sortchecking is still being developed. The present sortchecking is too lenient, which means that the user must perform some of the sortchecking manually, exercising care not to present certain kinds of ill-sorted terms to Penelope. Only record terms concerned us directly in verifying `Calendar_Utilities`.

Currently, Penelope has only one record sort, `AnyRecordSort`, and treats all record fields as naming components of this most general record sort. Since Ada record types are based on Larch/Ada record sorts, the user must never employ the same field name in two distinct record types, for otherwise, there would be an invalid overloading of the corresponding Larch record field name. We respected this limitation in defining record types.

Penelope effectively turns off sortchecking when treating record components, assigning them the most general sort `AnySort`, thus discarding information about their actual sorts. Consequently, sortchecking finds apparent ambiguities in perfectly unambiguous record terms. If a function `g` is overloaded, there may be no way for Penelope to determine the proper sort of `g(R.f)`

For instance, in `Calendar_Utilities`, we have the sort `fixnum` representing rational numbers. We overload multiplication, so that Penelope has both these signatures in its symbol table:

```
"*" : Int, Int -> Int
"*" : Int, fixnum -> fixnum
```

As a result, Penelope treats the following equation as ambiguous:

```
(i*m).numer = i*(m.numer)
```

The term on the left of the equation may have any sort, so the target sort of the second * might be either `Int` or `fixnum`. The sort of `m.numer` can be either `Int` or `fixnum`. Thus, Penelope cannot determine which signature to use.

To direct the sortchecker in these cases, we remove the apparent ambiguity, employing identity functions from given sorts to themselves. Applying an identity function informs the sortchecker of the sort of the argument term while preserving the term's mathematical value. We can unambiguously write the formula as follows:

```
(i*m).numer = Int_identity(i*(m.numer))
```

Now, the only choice of signature for * is the first signature

```
"*" : Int, Int -> Int
```

Summary of the Traits The full text of the Penelope verification output for `Calendar_Utilities` given in Appendix E begins with six traits that we wrote in order to carry out the verification. Here is a brief summary of each trait.

- Trait `Z`

  Many, but not all, facts of arithmetic are built into Penelope. Trait `Z` contains additional facts about division and modular arithmetic. This trait also has an explicit identity function on the integers, which is useful for removing ambiguities during sortchecking.

- Trait `fixed_point`

  The theory of fixed point numbers is not yet built into Penelope. We use this trait to provide needed facts.

- Trait `sort_names`

  In this trait, we give all of the names of sorts that we later use to define the vocabulary of the Larch/Ada specification. The Ada types of `Calendar_Utilities` are based on the sorts of this trait. Some sorts have a "good_" predicate that asserts that a given value within the sort is a

legitimate representation of a date and time. The "_tic_" functions represent the values of Ada attributes. The trait also defines miscellaneous arithmetic and sort-forcing functions.

We introduce the enumeration sort **century** in order to provide the vocabulary to document that the Ada function **Value_Of** provided in package **Calendar_Utilities** is correct only for argument dates in the twentieth century:

```
SORT century IS (twentieth, twentyfirst)
```

Without **century** terms, we would be unable to express mathematically that "02/27/55" might refer equally well to 1955 or 2055.

We also assign names, or aliases, to the sort **Int**. Penelope permits one or more aliases for a sort, which may be used to make the mathematics clearer and Penelope easier to use. Aliases are especially useful for clarity when several Ada types are based on the same mathematical sort.

In the mathematical tier of this trait, we define the integers only once. All integer types and subtypes are based on the single infinite sort **Int**. Each type determines a subset of **Int**, over which its values range.

Hence the years, months, etc., of **Calendar_Utilities** are all based on the sort **Int**. To declare the signatures of our mathematical functions clearly, it is helpful to use the following aliases:

```
SORT years          IS Int
SORT months         IS Int
SORT month_days     IS Int
SORT year_days      IS Int
SORT absolute_days  IS Int
SORT hours          IS Int
SORT minutes        IS Int
SORT seconds        IS Int
SORT milliseconds   IS Int
```

This signature declaration from trait **time_representations**

```
day_of_year: years, months, month_days -> year_days
```

shows clearly what the values in the arguments mean, whereas

```
day_of_year: Int, Int, Int -> Int
```

would be unenlightening.

There are many other aliases of this clarifying nature in **sort_names**. Separate from them is the alias

```
catu ALIASES Int
```

We will use this alias later when we introduce a very short, but unspecified, unit of time, the *catu* (Calendar Atomic Time Unit). **Catus** relate the abstract time values from the implementation of **Calendar.time** to the time values of **Calendar_Utilities.time**.

The abstract sort **Calendar.time_sort** may have a smaller granularity than the millisecond: we do not know. By choosing to work with the small but unspecified **catu**, we do not have to know. We require of the **catu** only that it be positive and of sufficiently small granularity to represent faithfully times expressed in either setting. We treat it as an unspecified positive integer constant in the remainder of the specification.

- Trait **conversion_factors**

  This trait defines the functions for converting from hours to seconds, days to milliseconds, etc.

- Trait **time_representations**

  This trait encapsulates the theory of the individual time representations. We concentrate on extracting information from time values of the sort **Time_sort** on which the type **Time** of **Calendar_Utilities** is based. For instance, we find the number of days in a given month in a given year through the function **days_in_month**.

  We also define predicates that indicate whether formal time values represent possible times: **good_year**, **good_month**, **good_month_day**, and so on. We use these predicates to define the predicate **good_time** on an entire **Time_sort** value

```
(the_year => 1990,
 the_month => 2,
 the_day => 29,
 the_hour => 12,
 the_minute => 34,
 the_second => 56,
 the_millisecond => 789)
```

  Since there is no February 29, 1990, this formal record value t does not represent an actual time. Accordingly,

$$\texttt{good\_time}(t) = \mathit{false}$$

  We can use these predicates to document that a given Ada subprogram will behave correctly only when its arguments represent possible dates and times.

- Trait `time_representation_conversions`

  This trait encapsulates the theory of converting from one time representation to another.

4.2.2 Detailed Specifications

Here we present and discuss the Larch/Ada specification of every subprogram in the package `Calendar_Utilities` obtained from NASA. The mathematics used in the specification is found in the traits discussed above in Section 4.2.1. The disambiguating comments after overloaded function names are the same as in Section 3.

The style of each specification is much the same as every other. After reading the first few, as well as the specification of `time_of` (Time), the reader may wish to proceed to the general discussion of the specifications in Section 4.3. The specification of `time_of` (Time) deserves attention because it is the first in which the Ada type to Larch sort correspondence is discussed in full depth.

Function `is_leap_year`

```
FUNCTION is_leap_year(the_year : IN year) RETURN boolean;
--| WHERE
--|       IN (good_year(the_year));
--|       RETURN is_leap_year(the_year);
--| END WHERE;
```

This function calculates whether or not the given year is a leap year. The `IN` annotation uses the predicate `good_year` to restrict the value of the argument `the_year` to the acceptable interval [1901, 2099]. The `RETURN` annotation specifies that the the returned `boolean` has value *true* if and only if `the_year` is a leap year, using the predicate `is_leap_year` to do so.

Note from the absence of global annotations that this function must have no side effects. Similarly, since there are no propagation annotations, this function must not terminate by propagating an exception.

Function `days_in` (Year)

```
FUNCTION days_in(the_year : IN year) RETURN year_day;
--| WHERE
--|       IN good_year(the_year);
--|       RETURN days_in_year(the_year);
--| END WHERE;
```

This function calculates the number of days in a given year. The `IN` annotation restricts the value of the argument `the_year` to the acceptable interval [1901, 2099]. The `RETURN` annotation specifies that function computes the number of days in the year. There are to be no side effects or propagated exceptions.

Function days_in (Month)

```
FUNCTION days_in(the_month : IN month;
  the_year : IN year) RETURN day;
--| WHERE
--|       IN good_month(the_month);
--|       IN good_year(the_year);
--|       RETURN days_in_month(the_month, the_year);
--| END WHERE;
```

This function calculates the number of days in a given month. The year must also be given because of February and leap years. The first **IN** condition constrains the value of **the_month** to lie in the interval [1, 12]. The second **IN** annotation restricts the value of the argument **the_year** to the acceptable interval [1901, 2099]. The **RETURN** annotation specifies that the number of days in the month is to be computed. There are to be no side effects or propagated exceptions.

Function month_of (Month Number)

```
FUNCTION month_of(the_month : IN month) RETURN month_name;
--| WHERE
--|       IN good_month(the_month);
--|       RETURN month_name_sort_tic_val((the_month-1));
--| END WHERE;
```

This function converts numerical months to the corresponding months within the enumeration type **(January, February, ..., December)**. The **IN** annotation restricts the numerical month to the interval [1, 12]. The **RETURN** annotation specifies the enumeration value corresponding to the numerical month.

The language of this specification requires some explanation. First, the Larch/Ada function

```
month_name_sort_tic_val
```

represents the values of the Ada attribute function

```
month_name'val
```

The name of the Larch/Ada function is intended to suggest that it takes values in sort **month_name_sort** on which Ada type **month_name** is based. The "**_tic_**" is used to represent the tic (apostrophe), which is not acceptable as part of Larch/Ada function names. The definition of **month_name_sort_tic_val** is given in trait **sort_names**, found in Appendix E.

Second, months of the calendar are numbered from 1, while Ada's enumeration types are numbered from 0. In Ada, **'pos(January)** has value 0, not 1. Subtracting 1 from **The_Month** is the necessary adjustment.

There are to be no side effects or propagated exceptions.

Function month_of (Month Name)

```
FUNCTION month_of(the_month : IN month_name) RETURN month;
--| WHERE
--|       RETURN (month_name_sort_tic_pos(the_month)+1);
--| END WHERE;
```

Months in the enumeration type (**January, February,..., December**) are converted to the corresponding numerical months. The **RETURN** annotation specifies the enumeration value corresponding to the numerical month.

The language of this specification requires some explanation. The Larch/Ada function **month_name_sort_tic_pos** represents the values taken by the Ada attribute function **month_name'pos**. Subtracting 1 from **The_Month** is necessary to adjust from the 0-based numbering of Ada enumeration types to the ordinary 1-based numbering of months. For more information, please read the exposition of the preceding Ada function.

There are to be no side effects or propagated exceptions.

Function day_of (Year, Day)

```
FUNCTION day_of(the_year : IN year;
  the_day : IN year_day) RETURN day_name;
--| WHERE
--|       IN good_year_and_day(the_year, the_day);
--|       RETURN day_name_of(the_year, the_day);
--| END WHERE;
```

This function calculates the day of the week for a given date. The **IN** annotation specifies that the year and day together constitute a valid date. The **RETURN** annotation specifies that the proper day of the week is calculated. There are to be no side effects or propagated exceptions.

Function day_of (Time)

```
FUNCTION day_of(the_time : IN time) RETURN year_day;
--| WHERE
--|       IN good_time(the_time);
--|       RETURN day_of_year(the_time.the_year,
  the_time.the_month, the_time.the_day);
--| END WHERE;
```

This function differs from the previous **day_of** function only in the representation of the given date. The **IN** annotation specifies that **the_time** is a good representation of a date and time. The **RETURN** annotation specifies that the proper day of the week is calculated. There are to be no side effects or propagated exceptions.

Function time_of (Year, Day)

```
FUNCTION time_of(the_year : IN year;
  the_day : IN year_day) RETURN time;
--| WHERE
--|       IN good_year_and_day(the_year, the_day);
--|       RETURN t SUCH THAT (
            t.the_year=the_year
          AND
            t.the_month=month_of(the_year, the_day)
          AND
            t.the_day=day_of_month(the_year, the_day)
          AND
            t.the_hour=0
          AND
            t.the_minute=0
          AND
            t.the_second=0
          AND
           t.the_millisecond=0);
--| END WHERE;
```

This function converts the date from representation by a Year-Day pair to **time** representation. The **IN** annotation specifies that the Year-Day pair must actually represent a date. The **RETURN** annotation specifies that the returned **time** value will represent the same date, at the instant it begins. There are to be no side effects or propagated exceptions.

Function period_of

```
FUNCTION period_of(the_time : IN time) RETURN period;
--| WHERE
--|       IN good_time(the_time);
--|       RETURN (IF (the_time.the_hour<12) THEN am  ELSE pm);
--| END WHERE;
```

This function computes whether a given time of day is before noon or not. The **IN** annotation specifies that **the_time** is a good representation of a date and time. The **RETURN** annotation specifies that **am** is to be returned for times before noon, and that **pm** is to be returned for times from noon onward. There are to be no side effects or propagated exceptions.

Function time_of (Time)

```
FUNCTION time_of(the_time : IN time) RETURN calendar.time;
```

```
--| WHERE
--|       IN good_time(the_time);
--|       RETURN time_to_cal_time(the_time);
--| END WHERE;
```

This function converts **time** representations to **calendar.time** representations of time and date. Before we discuss its annotations, recall that the value of the Ada variable **the_time** is in Larch sort **time_sort** since that is the sort on which the Ada type **time** is based.

The **IN** annotation specifies that **the_time** is a good representation of a date and time. The value of **the_time** belongs to Larch sort **time_sort**. Applying the mathematical function **time_to_cal_time** to this value yields the corresponding equivalent value in the sort **calendar_time_sort**. The **RETURN** annotation specifies that this equivalent value is the value returned by the Ada function.

There are to be no side effects or propagated exceptions.

Function time_of (Calendar.Time)

```
FUNCTION time_of(the_time : IN calendar.time) RETURN time;
--| WHERE
--|       RETURN cal_time_to_time(the_time);
--| END WHERE;
```

This function differs from the previous **time_of** function basically in that the representations of time are reversed: the conversion proceeds in the opposite direction. The **IN** annotation is implicitly **TRUE**: all **calendar.time** values are valid. The **RETURN** annotation specifies that value returned is the equivalent, in the **time** representation, of the original value, which is in the **calendar.time** representation. There are to be no side effects or propagated exceptions.

Function time_image_of

```
function time_image_of (the_time  : in time;
                        time_form : in time_format)
return string;
--| WHERE
--|       IN good_time(the_time);
--|       RETURN time_to_time_string(the_time, time_form);
--| END WHERE;
```

This function extracts the time of day from the given **time** representation and converts it to a **string** representation in the given format. The **IN** annotation specifies that **the_time** is a good representation of a date and time. The **RETURN** annotation specifies that the returned **string** represents the input time in the chosen time string format. There are to be no side effects or propagated exceptions.

Function date_image_of

```
function date_image_of (the_time  : in time;
                        date_form : in date_format)
return string;
--| WHERE
--|      IN good_time(the_time);
--|      RETURN time_to_date_string(the_time, date_form);
--| END WHERE;
```

This function differs from the above in that it converts the extracted date, not time, to the chosen string representation. The **IN** annotation specifies that **the_time** is a good representation of a date and time. The **RETURN** annotation specifies that the returned **string** represents the input date in the chosen date string format. There are to be no side effects or propagated exceptions.

Function value_of (Date/Time Strings)

```
function value_of (the_date  : in string;
                   the_time  : in string;
                   date_form : in date_format;
                   time_form : in time_format)
return time;
--| WHERE
--|      RETURN strings_to_time(    the_date,
--|                                 the_time,
--|                                 date_form,
--|                                 time_form,
--|                                 twentieth
--|                             );
--| RAISE lexical_error <=>
--|   (NOT well_formed_date_string(the_date, date_form))
--| OR
--|   (NOT well_formed_time_string(the_time, time_form));
--| END WHERE;
```

This function converts date and time from **string** representation to **time** representation. The absence of an **IN** annotation indicates that any input is acceptable. The **RETURN** annotation specifies that the returned **time** value is to be equivalent to the representation of time and date in the given input formats. Notice that we have explicitly documented that the code treats every two-digit year as belonging to the twentieth century. The **RAISE** annotation specifies that the function will raise **lexical_error** if and only if the string input is invalid. There are to be no side effects.

Function duration_of

```
FUNCTION duration_of(the_interval : IN interval)
  RETURN builtin.duration;
--| WHERE
--|       RETURN interval_to_duration(the_interval);
--|       RETURN result SUCH THAT good_duration(result);
--| END WHERE;
```

This function converts time in **interval** representation to time in **duration** representation. The absence of an **IN** annotation indicates that any input is acceptable. The first **RETURN** annotation specifies that the returned **duration** value is to be equivalent to the given **interval** value. The second **RETURN** annotation is a convenience for verification. Without it, every time that **duration_of** is invoked, the user is likely to have to prove the predicate **good_duration** of the returned value. With it, the user need only prove the predicate once, within the verification of **duration_of** itself. Therever after, the predicate will automatically apply to the returned result, and need not be reproved. There are to be no side effects or propagated exceptions.

Function interval_of

```
FUNCTION interval_of(the_duration : IN builtin.duration) RETURN
 interval;
--| WHERE
--|       RETURN duration_to_interval(the_duration);
--| END WHERE;
```

This function is the inverse of the preceding function. No **IN** conditions need to be placed on **the_duration**, since all **duration** values are valid. The **RETURN** annotation specifies that the returned value is the **interval** equivalent of the input value. There are to be no side effects or propagated exceptions.

```
function image_of (the_interval : in interval) return string;
--| WHERE
--|       RETURN interval_to_string(the_interval);
--| END WHERE;
```

This function returns the **string** representation of the time in **interval** form. There is no **IN** annotation because all **interval** values are valid. The **RETURN** annotation specifies that the returned value is the **string** equivalent of the **interval** input. There are to be no side effects or propagated exceptions.

Function value_of (Interval String)

```
function value_of (the_interval : in string) return interval;
--| WHERE
--|       RETURN string_to_interval(the_interval);
--|       RAISE lexical_error <=>
--|         NOT well_formed_interval_string(the_interval);
--| END WHERE;
```

This function converts time intervals in **string** representation to time intervals in **interval** representation. All inputs are acceptable: there is no explicit **IN** annotation. The **RETURN** annotation specifies that the returned **interval** value is to be the equivalent of the input **string** interval value. There are to be no side effects or propagated exceptions.

4.3 Discussion of the Specification

In this section, we first explain our approach for formally specifying existing software. We then describe the problems with the `Calendar_Utilities` package that were revealed when we specified the package.

4.3.1 Formally Specifying and Verifying Existing Software

Ideally, we formally specify software before it is written, and write the software in tandem with verifying it. Formal methods are best applied to a program when it is under development. In practice, however, we often specify and verify existing software, that is, software written previously, independently of us. Although verifying existing software applies only part of the strength of formal methods, we still can uncover problems with software.

When verifying existing software without an original specification, we cannot absolutely know what the software was meant to do. We make assumptions in writing the specification and we document those assumptions. We then verify the software with respect to our specifications.

In the case of the `Calendar_Utilities` package, we used the following approach to specify the software. First, for reasonable inputs, we specify that the output be what people would expect. For example, a reasonable input for a date could be March 2, 1991, and people would expect a conversion of that date to be 03/02/91 (in terms of the string representation formats in the program). Second, we specified that the input must be reasonable. The functions we were given accept unreasonable inputs without complaint. For example, the function

```
function Time_Of (The_Time  : in Time) return Calendar.Time;
```

as written accepts impossible dates (e.g., March 31) and returns invalid values for type `Calendar.Time` without raising an exception. If the invalid values are then passed as arguments to operations of package `Calendar`, an exception will be raised.

We used restrictive `IN` annotations to show that the `Time_Of` can be trusted only on certain inputs.

We verify the software with respect to our assumptions and specifications. The demonstrated correctness of a program assures that the program satisfies our mathematical statement of its behavior; a formally based judgement of incorrectness assures that the program does not satisfy our description of its intended behavior.

4.3.2 Discussion of the Software

For most of the `Calendar_Utilities` subprograms, the Larch/Ada specifications document the clearly intended design of the software. Informal inspection assured us that the design was well implemented. However, the entire package was not problem-free. We found and documented subprograms that do not

guard against unsafe arguments, that is, arguments that lead to unexpected results. Of greater concern, we also found dubious features and outright errors in the software.

Software with No Apparent Problems Functions such as `Is_Leap_Year` presented no difficulties. The `IN` annotation is practically trivial: the year lies in the intended range. There are no `RAISE` annotations. We informally determined that this function will behave as intended for all permissible inputs.

Software with Minor Problems We found that some functions, such as `Day_Of` (Time) and `Time_Of`, do not guard against unsafe arguments. We therefore specify, for example, that the `Time` value passed to `Day_Of` must not represent such impossible dates as September 31. The `IN` annotation defines the acceptable input values.

Contrast this with function `Value_Of` (Date/Time Strings), which does guard against unsafe values by raising exceptions. We document what these unsafe arguments are with the `RAISE` annotation. The `IN` annotation can be the trivial condition `TRUE`, because `Value_Of` can be called with any input.

Problematic Software We begin by describing some shortcomings in the software that we found in the process of performing the specification. Related to the above-mentioned numerical representations of impossible dates, there is unfaithful extraction of information represented in strings.

In the function `Value_Of` (Date/Time Strings), which takes a string and returns a `Time` record, the year from "02/27/55" becomes 1955. This assumption that all two-digit year dates represent years in the twentieth century is dubious, particularly for software written so near the end of the century. The software should have a century parameter to allow the user to select the century for the date. (We have such an argument in our specification function, but its role is purely documentary, not restrictive. The software, not merely the specification, needs to take centuries into account.)

We found this dubious feature while attempting to define an inverse to the original mathematical record-to-string conversion: the inverse did not exist because century information was being thrown away. We therefore defined the sort `century` to carry the necessary information.

After completing the specification, we systematically compared the software with the description of what it should do, though this comparison was of course informal. We marked a number of problems in the software for closer examination, the most salient three are listed below. We tested these problems before the verification phase and determined that two of the problems were indeed errors, while the third was a dubious feature of the software.

We had one difficulty with testing that requires explanation. Of the library units that Calendar_Utilities depends upon, several were available to us only in

incomplete form and were not compilable. To run our test cases after we identified potential errors, we removed sections of the software and wrote a handful of lines so that the necessary units would compile. None of our alterations affected the software in `Calendar_Utilities` undergoing formal analysis.

First Problem In the body of `Calendar_Utilities`, `Value_Of` (Date/Time Strings) contains the following lines of software (some of the initial lines are omitted).

```
   function Value_Of (The_Date  : in String;
                      The_Time  : in String;
                      Date_Form : in Date_Format := Full;
                      Time_Form : in Time_Format := Full)
    return Time is
       Result      : Time;
       Left_Index  : Positive;
       Right_Index : Positive;
   begin
--     ...

       case Time_Form is
          when Full     =>
              Right_Index :=
                String_Utilities.Location_Of
                  (The_Character => Blank,
                   In_The_String =>
                     The_Time(Left_Index ..The_Time'Last)
                  );
              Result.The_Millisecond :=
                Millisecond(Natural_Utilities.Value_Of
                  (The_Time
-- (1)***
                     (Left_Index .. (Right_Index - 1))));
              Left_Index := Right_Index + 1;
              if Period'Value(The_Time
                (Left_Index .. The_Time'Last)) = PM then
                  if Result.The_Hour /= Noon then
                      Result.The_Hour :=
                        Result.The_Hour + Noon;
                  end if;
              end if;
          when Military =>
              Result.The_Millisecond :=
                Millisecond(Natural_Utilities.Value_Of
```

```
-- (2)***
                    (The_Time(Left_Index .. The_Time'Last)));
        end case;
        return Result;
    exception
        when Constraint_Error =>
            raise Lexical_Error;
    end Value_Of;
```

`Value_Of` takes two string representations of time (e.g., "FEBRUARY 27, 1955" and "01:21:06:30 PM") and extracts the corresponding numerical components to build a record representation. The assignments indicated by `(1)` and `(2)` in the code above take hundredths of a second (the "30" from the second string above) and insert the millisecond component into the record. There is no multiplication by 10, so the milliseconds in our example are recorded as 30, instead of 300. Re-conversion back to a string, using the correct function `Time_Image_Of` (employing the expected division by 10) produces the different string "01:21:06:03 PM".

Second Problem In the body of package `Calendar_Utilities`, the function `Duration_Of` (which immediately follows `Value_Of`) is as follows.

```
function Duration_Of (The_Interval : in Interval)
return Duration is
begin
  return (  Duration(Integer(The_Interval.Elapsed_Days)
          * Seconds_Per_Day) +
            Duration(Integer(The_Interval.Elapsed_Hours)
          * Seconds_Per_Hour) +
            Duration(Integer(The_Interval.Elapsed_Minutes)
          * Seconds_Per_Minute) +
          Duration(The_Interval.Elapsed_Seconds) +
-- (1)***
          Duration(The_Interval.Elapsed_Milliseconds / 1000));
end Duration_Of;
```

This function takes a record representation of time and returns a numerical fixed-point representation. In line `(1)`, an integral number of milliseconds is intended to be converted into a fractional number of seconds. Thus 276 milliseconds should convert to 0.276 seconds. The conversion to fixed-point type `Duration` occurs, however, *after* integer division. Millisecond values are between 0 and 999 and hence will always produce 0 when divided by 1000. Thus, `Duration_Of` wrongly converts any `Interval` record into a whole number of seconds.

Third Problem Consider again the function `Value_Of` (Date/Time Strings).

```
    function Value_Of (The_Date  : in String;
                       The_Time  : in String;
                       Date_Form : in Date_Format := Full;
                       Time_Form : in Time_Format := Full)
    return Time is
        Result      : Time;
        Left_Index  : Positive;
        Right_Index : Positive;
    begin
        case Date_Form is
           when Full           =>
               ...
           when Month_Day_Year =>
               ...
-- (1)***
               Result.The_Day := Day(Natural_Utilities.Value_Of
                 (The_Date (Left_Index .. (Right_Index - 1))));
               ...
        end case;
    ...
    end Value_Of;
```

In line (1), which occurs in the first case statement of the software, the day of the month is selected from the date string. Thus if the string is "FEBRUARY 27, 1955", we wish to select the "27" and then convert it to the integer value of 27. An Ada slice is taken, using

```
(Left_Index .. (Right_Index - 1))
```

In our example, the Ada `'first` and `'last` values of the slice are 10 and 11, respectively. This fairly arbitrary slice is then passed to the separate package `Natural_Utilities` for conversion to a numerical value. A better approach might be to "normalize" the slice to one whose range was 1..2, rendering the software robust across differing implementations of the `Natural_Utilities` package. In fact, the package `Natural_Utilities` that came with the software was written assuming that strings began at position 1, and an exception was raised when we ran `Value_Of` on this example.

4.4 Verification

In Section 2.3, we described formal verification in Penelope in general terms. In this section, we will describe the verification of `Calendar_Utilities` in detail.

First, we place qualifications on our results. Second, we comment briefly on the subprograms that we fully verified. Third, we discuss the errors we detected. Finally, we summarize the results.

4.4.1 Qualifications of the Results

When we state that software has been verified as correct using Penelope, we must consider three qualifications. First, we are working within Penelope's program semantics, which do not model all the possible behaviors of an Ada program. An Ada program correct in the Penelope semantics may still have certain unacceptable behaviors. Second, correctness is verified with respect to a specification that may not exactly reflect the developer's intention. Third, the correctness is verified for a modified version of the `Calendar_Utilities` code. Because Penelope is still being developed, we had to modify the code to be in the subset of Ada that Penelope currently supports. Some modifications were major changes to the code, and others were minor changes to accommodate the limitations of Penelope.

Correctness Within Penelope's Program Semantics Our notion of correctness is termed *partial correctness*. Partial correctness means that a program is correct if whenever it is started in a state that satisfies all of its initial conditions and the program also halts, it halts in a state that satisfies all of its final conditions. The program may, however, fail to halt. A program that loops forever will be correct by our semantics.

Fortunately, failure to halt is not a concern in this effort, since the only iteration constructs in the verified code are `FOR` loops, which have been transformed into equivalent `WHILE` loops. The upper bounds have predefined maximum values. There are no recursively defined routines.

Secondly, our notion of correctness excludes the possibility of numeric overflow or storage error. The formal basis of Penelope deliberately omits them. Penelope verification is predicated upon programs not raising those exceptions.

For a fuller description and discussion of the meaning of a complete Penelope verification, see Guaspari [9].

Specifications The software is proved to meet specifications that we supplied. If our specifications are not the proper ones, the verification of the software is irrelevant. For the verification to be relevant, we must reflect the intent of the developer.

We must also define the logical theory correctly. If the vocabulary from the theory does not have its intended meaning, or worse yet, if the theory is

self-contradictory, the verification is suspect. Furthermore, the asserted lemmas may not actually follow from the axioms. Currently, the only safeguard against these defects is the expertise of the verifier. Efforts are underway to provide automated assistance to users of Penelope in constructing theories.

Major Changes to the Code There are four types representing time in package `Calendar_Utilities`. There is the private type `Time` from the package `Standard.Calendar`, the record type `Time` of the package `Calendar_Utilities`, the record type `Interval` of the package `Calendar_Utilities`, and the built-in fixed-point type `Duration`. The first three types presented no problems, but we had to make major changes to the code to treat the last type.

The Ada Reference Manual does not specify a concrete implementation for the private type `Time`, nor does the concrete type matter so long as we are able to represent values of types `year_number`, `month_number`, `day_number` and `day_duration` as objects of type `time` in a manner that respects the operations. We must be able to convert to type `time`, add, subtract, compare and subsequently convert back to the correct values by means of the functions in the package. Beyond this, the particulars of type `time` are irrelevant.

We used `integer` as the concrete type for `Time`: it satisfies all the necessary conditions. We conceive of time as integer multiples of a fundamental unit, the `catu` (calendar atomic time unit), of which all of our other time units are integer multiples. The traits describe the theory for converting between all our discrete representations of time, all based on the `catu`.

Everything would be completely straightforward if it were not for the fixed-point type `duration`. ORA has addressed the theory of real number types [11], and real number verification will be melded with Penelope in Penelope/Ariel, schedule for release in 1992. In the meantime, we must give some treatment of the fixed-point type `Duration`.

We changed the Ada declaration of `Duration` to be a record type pairing numerator and denominator; we based all fixed-point numbers and operations on the rational numbers under addition and multiplication. In doing so, we assumed that the error inherent in rounding, addition, assignment, and so on is too small to affect the basic correctness of our results.

Because fixed point operations in Ada are not exact, while our rational operators are, a critic could reasonably argue that our verification should be performed again in Penelope/Ariel, which supports real number types. We feel, however, that our treatment of fixed-point types was sufficient for us to certify the correctness of some subprograms and to detect the incorrectness of others. The differences between rational and fixed-point operations should be negligible. We would expect no surprises under subsequent re-verification.

Accommodating Limitations of Penelope To accommodate the limitations of our current editor, we made minor changes to the code and performed

some tasks manually. The changes we made to the code were innocuous. These changes included translating the Ada text to lower case, using the fully qualified forms of several names, converting from one type to another with function calls instead of with explicit type conversions, translating **FOR** loops to equivalent **WHILE** loops. In addition, we altered the form of certain procedure calls in order to satisfy Penelope's conservative restrictions against potential aliasing. We removed the declarations of exceptions because Penelope does not yet support their Ada declarations; they are, however, implicitly declared by mention in Larch/Ada. Finally, we changed named parameter associations to positional parameter associations. None of these changes affected the Ada semantics.

We overcame other limitations by doing manually what Penelope does not yet do automatically. We explicitly coded range constraint checks. We type-checked Ada expressions by hand. We simulated Larch/Ada attribute functions by functions we defined ourselves. We simulated Larch/Ada aggregates similarly. All of this work was performed in accordance with our mathematical theory of verification.

The details of these changes and manual tasks are described below.

- Restriction to lower case

  Ada identifiers and reserved words are case-independent (RM 2.3[3], RM 2.9[4]). The Ada object `obj` may be denoted by "OBJ", "Obj", "obj", etc. The Ada programmer is free to use any mixture of case desired for object identifiers.

  Larch, however, is case sensitive [10]. The Larch/Ada terms "OBJ", "Obj", and "obj" are, thus, all distinct linguistic entities—only one of these terms can canonically represent the value of the Ada object. Early in the development of Penelope, we made the all-lower-case form canonical.

  So as to maintain a clear relationship between Ada objects and the Larch terms that represent their values, we converted all the identifiers appearing in **Calendar_Utilities** to lower case. This change did not affect the Ada semantics.

- Fully expanded expressions

  Penelope does not yet support the **USE** context clause. When referring to types and objects of **WITH**ed packages, it was necessary to write their fully expanded names. Thus, we needed to change the references to type **year_number** of package **calendar** to read "**calendar.year_number**". In the same vein, we declared and **WITH**ed a package **builtin** so that we could refer to the built-in type **duration**.

- Constraints and ranges

  Penelope does not yet support range constraints. The integer type definitions in the original code have therefore been transformed to declarations

of unconstrained **new integer** types. Larch/Ada predicates for the sorts corresponding to these types have been defined so that we can code range checking explicitly.

The names of these functions are the names of the corresponding types prefixed with "**good_**". Their definitions are taken from the bounds of the range in the original Ada text of **Calendar_Utilities**. We used them systematically in the specifications of programs that have formal parameters of the original range type, and as conditions on axioms and lemmas that have variables of the corresponding sort. Doing so ensured that range checking is always part of the correctness condition of the verified software.

For instance, the explicit Larch/Ada IN annotation

```
IN good_month(the_month);
```

takes the place of the implicit constraint check

$$1 \leq \textbf{the\_month} \leq 12$$

for the Ada type **Month**.

- Derived types

  Penelope does not yet fully support derived types in Ada. The type mark[11] of a derived type is today treated as an alias for the parent type instead of as an isomorphic copy. This means that Penelope's typechecking of Ada is somewhat too lenient, as it does not guard against improperly mixing parent and derived types in an Ada expression. It was part of our task as verifiers to take such precautions manually.

- Type conversion

  Type conversions are not yet supported by Penelope. Its typechecker does, however, support the restriction against using names of types as function names. Where explicit type conversions appear in the original Ada code, therefore, they are rejected by Penelope as illegal function calls. To overcome these limitations, we have replaced all explicit type conversions with calls to actual functions whose specified effect is to perform the desired type coercions.

  The name of a coercion function follows this convention: the name of the target type has the suffix "**_ize**". Thus a function that coerces to type **integer** is named "**integer_ize**"[12]. Since the coercion functions

[11] See the Ada RM 3.3.2:2-3, 3.4

[12] Ada overloading permits as many identically-named functions as required, one for each type to be coerced. The typechecker determines which function is intended from the type of the argument.

are purely auxiliary, no bodies for them are given, only Ada specifications. Their Larch/Ada specifications provide the predicate transformers with all of the information that is required to perform the necessary type coercions.

That constrained types and ranges are not yet supported introduces further complications. There is currently no distinction in Penelope between the types **integer**, **natural** and **positive**; consequently, are all the same type: the conversion between them is apparent, not real. The "**_ize**" functions for the numeric types thus actually all take **integer** arguments and return **integer** results under another name.

We have these pseudo-conversions in the code for two reasons. First, we wish to change the text as little as possible. The change in going from **natural(obj)** to **natural_ize(obj)** is smaller than in going from **natural(obj)** to **obj**. Second, we express the constraints that must hold for the object as Larch/Ada **IN** specifications on the "**_ize**" functions. Thus, we specify that the input value to **natural_ize** must be non-negative. In fact, we explicitly state all of the constraints on objects that Penelope will later infer from their types.

- Attribute functions

  Penelope does not yet support the automatic generation of attribute functions, nor is the "tic" (i.e., the character "'") notation for them acceptable syntax. We replaced each function of the form **<type_mark>'<attribute>** in the original code by a function named **<type_mark>_tic_<attribute>**. We have added Ada specifications for these functions; the new Ada specifications are identical to those defined for the corresponding attribute functions. These pseudo-attribute functions have been annotated with Larch/Ada specifications that correctly describe the semantics of the attribute function by means of axioms that have been entered manually into the traits. The entire process is well defined and capable of automation.

- Aggregates

  Penelope does not yet support aggregate expressions. We circumvented this limitation in each case by declaring a temporary object of the result type, assigning the appropriate values to its fields, and returning the object thus constructed.

  Penelope will someday have aggregate-like constructs in Larch/Ada. Even without them, however, we have full expressiveness in Larch/Ada to name any (finite) array value. The array-update notation

  ```
  Array[index => value]
  ```

  denotes the array having value **value** at index **index** and agreeing with **Array** at every other index. Repeated application of the array-update

notation can be used to denote an array having a chosen value at each index.

To suggest the aggregate-like notation that we will someday implement, we declared constant functions on array sorts and updated all of the components of these undetermined constants whenever we wished to name a particular array value. One example is the term denoting the array of days of the months in an ordinary year:

```
month_days_tic[1=>31][ 2=>28][ 3=>31][ 4=>30]
               [5=>31][ 6=>30][ 7=>31][ 8=>31]
               [9=>30][10=>31][11=>30][12=>31]
```

- `FOR` loops

  `FOR` loops are not yet included in the subset of Ada for which Penelope has verification rules. `FOR` loops in the code have been replaced by `WHILE` loops whose test is that the index variable has reached the upper bound. The index variable is initialized to the lower bound by a preceding assignment statement and is incremented by an assignment statement at the end of the loop body.

- Aliasing

  Penelope has very conservative checks against improper aliasing; these checks reject subprogram calls which would result in certain kinds of program errors[13], but may also reject subprogram calls that are error-free. In particular, Penelope does not allow two components from the same record object to appear as actuals for **out** or **in out** formal parameters—for example, the call

```
calendar.split(result.the_year,
               result.the_month,
               result.the_day)
```

  To accommodate the checks, we rewrote the call slightly. The modified code calls `calendar.split` with temporaries in place of the record components and assigns each of the temporaries to the corresponding component after the procedure call.

- Exceptions

  The circumstances under which `Constraint_Error` would be raised by the application of `day_name_tic_val` are specified in the Larch/Ada annotation of that function.

[13] Technically, certain kinds of erroneous executions and incorrect order dependences.

Penelope does not yet support Ada restrictions on declaration of exceptions. Instead, Penelope considers an exception to be implicitly introduced at its first occurrence in propagation annotations. We removed exception declarations to perform this verification. After this change, Penelope is, in effect, more lenient than Ada in the declaration of exceptions. We therefore ensured that we did not introduce new exceptions.

- Actual Parameter lists

  Penelope does not yet support named parameter association. We changed occurrences in the original code to positional parameter association. This change affects only the convenience of the caller, not the correctness of the subprogram.

4.4.2 Results

Verified Functions We fully verified the functions `Is_Leap_Year`, `Days_In` (both `Month` and `Year` versions), `Month_Of` (both `Number` and `Name` versions), `Day_Of` (`Time` version), `Time_Of` (`Year, Day` version) and `Period_Of`.

Detected Errors The function `Duration_Of` contains a programming error that was detected first in the specification phase and is discussed in Section 4.3. In this verification phase, the error was highlighted by completing the proofs for all of the VCs of the program except for one, which can be seen to closely correspond to the incorrect expression.

The goal of the VC, paraphrased, says that the value held by an integer variable (`The_Interval.Elapsed_Milliseconds`) is unchanged when it is successively divided, then multiplied by 1000 (using integer arithmetic):

```
(the_interval.elapsed_milliseconds/1000)*1000
=
the_interval.elapsed_milliseconds
```

A well-trained verifier may respond to this clearly unprovable goal by recognizing its relation to the `RETURN` condition of the specification and pinpointing the line of code with the offending expression.

Two further errors were actually discovered during the verification process. Both errors occur in the function `Day_Of` (Year, Day). We detected them by realizing that we could not prove the VC generated for the loop invariant. The two remaining incomplete subproofs correspond to the two errors.

The function

```
Day_of(the_year: year; the_day: year_day) return day_name;
```

returns the name day of the week on which day number `the_day` falls in the year `the_year`. It proceeds by first executing a loop to determine the day of

the week on which **the_year** begins, and then shifting that day of the week by **the_day** (modulo 7). Each step contains an error.

The loop counter **Index** runs from **year'first+1** to **the_year**, and each iteration of the loop should advance the **day_name** stored in **result** by either one day (if **Index** is not a leap year) or two days (if it is). The VC for the loop, however, is this:

```
(IF is_leap_year(the_year)
  THEN (good_day_name(day_name_sort_increment(result, 2))
    ->
    (day_name_sort_increment(result, 2)
      =
      day_name_of(index, 1)))
  ELSE (good_day_name(day_name_sort_increment(result, 1))
    ->
    (day_name_sort_increment(result, 1)
      =
      day_name_of(index, 1))))
```

This is unprovable, because the **IF** assertion is branching, not on the value of **index**, but on the value of **the_year**. On checking the **IF** statement in the body of the loop, we find that it incorrectly tests the loop bound instead of the loop index; that is, **Is_Leap_Year(The_Year)**, instead of **Is_Leap_Year(Index)**.

The problem with the second step is that it attempts to calculate the shift as follows:

```
Natural((the_day mod 7) - 1)
```

If the value of **the_day mod 7** is zero, the type conversion to **Natural** will raise an exception. The shift should instead be calculated by

```
Natural((the_day - 1) mod 7)
```

This error was discovered by considering the VC

```
0 < (the_day MOD 7)

AND

( good_day_name(
    day_name_sort_increment(result, (the_day MOD 7) - 1)
    )
->
  ( day_name_sort_increment(result, (the_day MOD 7) - 1)
  =
   day_name_of(the_year, the_day)
  )
)
```

The assertion `0 < (the_day MOD 7)` is simply not true, and this directed us immediately to the mistaken calculation of the shift. (Note the origin of `0 < (the_day MOD 7)`: it is precisely what must hold in order to guarantee that the unwanted constraint error is not raised.)

The most intriguing thing about these last two errors is that they will not invariably lead to incorrect output from the function. Though we will not prove it here, the function will in fact return the correct value for the years 1984-87, as long as the value of the input parameter **The_Day** is not congruent to 0 **mod** 7. There is at least the possibility that the existence of a problem could go undetected through some moderately extensive, but unfortunately chosen, test cases.

Each of the errors described in this section is a local mistake and can be fixed with a small change to the program. The near-completeness of the corresponding VCs clearly shows that the resulting programs could then be fully verified.

4.4.3 Summary of Results

We specified all eighteen functions of the package `Calendar_Utilities` using Larch/Ada. Of these, we selected thirteen for formal verification. We treated eleven of the thirteen completely.

Of the eleven we treated formally, we verified eight and detected errors in three others. Seven functions were not treated formally.

Our results are in the table below. When two functions share a common name, we comment on their arguments enough to distinguish them.

Function	Comments	Verified	Incorrect	Not Treated
`Is_Leap_Year`		•		
`Days_In`	`(Year)`	•		
`Days_In`	`(Month)`	•		
`Month_Of`	`(Month Number)`	•		
`Month_Of`	`(Month Name)`	•		
`Day_Of`	`(Year, Day)`		•	
`Day_Of`	`(Time)`	•		
`Time_Of`	`(Year, Day)`	•		
`Period_Of`		•		
`Time_Of`	`(Time)`		•	
`Time_Of`	`(Calendar.Time)`			•
`Time_Image_Of`				•
`Date_Image_Of`				•
`Value_Of`	`(Date/Time Strings)`			•
`Duration_Of`			•	
`Interval_Of`				•
`Image_Of`				•
`Value_Of`	`(Interval String)`			•

5 Conclusions

This demonstration of the power and practicability of formal methods in Penelope was a success. Formal specification in Penelope was useful for analyzing the software, both informally and formally. Formal verification gave greater confidence in the correctness of most of the package. Where the software was incorrect, the errors detected by Penelope could have gone unnoticed in testing.

This verification effort was the first to use a non-built-in data type extensively in Penelope. Without the automatic simplification that is provided for the built-in types, we found ourselves spending far too much of our time manually applying simplification directives. Future directions for Penelope should include consideration of automatic simplification within user-supplied theories.

Library-based verification would have been greatly advantageous. Because of the sheer bulk of the Ada code within a single buffer, Penelope ran very slowly. This problem lends support to our conclusions, drawn before this demonstration, that library-based verification is a necessity and validates our efforts in that direction.

Penelope is a useful prototype today for developing correct software. As ORA continues to enhance Penelope, it will become a generally useful tool.

Appendix A: Related Work

The field of formal verification began with the work of Hoare [12, 13]. Dijkstra and Gries went on to develop and advocate a distinctive style of verification based on Hoare's ideas. A thorough basic introduction to Dijkstra-Gries style verification can be found in Gries [6]. The same material is covered more fully in Dijkstra [3].

A serious obstacle to performing Dijkstra-Gries style verification by hand is the crushing work of bookkeeping in verifying actual programs (as opposed to textbook exercises). Penelope [8] is an automated assistant that manages the details of predicate transformation and proof checking, freeing the user to perform the conceptual work of verification.

Other computerized verification systems include EHDM [2], Gypsy [5], ORA Ottawa's m-EVES [1], and SDVS [15].

The Larch [10] system of mathematical specification separates mathematical concerns from programming concerns. Significant research has investigated how best to specify programs in the Larch system [17].

Appendix B: Glossary

Assertion — A mathematical statement about a program state. To make an assertion is to claim that a certain predicate is true.

Floyd-Hoare Triple — The triple $\{initial\}\ S\ \{final\}$ symbolizes that software S transforms state *initial* into state *final*. The Floyd-Hoare triple is basic to the mathematics of program semantics.

Precondition — A predicate that must hold on the program state before a statement executes if the statement is to bring about its intended effect.

Predicate — A logical formula such as $x > 17$; a mathematical expression that is either true or false when its variables are assigned values.

Program Semantics — Loosely speaking, the behavior of software. Program semantics treats the behavior of a program as its meaning.

Sort — The logical category of a mathematical object, analogous to a data type. The sort of the mathematical integers is `Int`.

Specification — A user-supplied predicate describing the conditions that are to hold at various points in the execution of a program.

- Input specifications describe the initial state of the program; that is, what must be true before the program executes.
- Output specifications describe the final state of the program. The circumstances of both normal and exceptional termination can be specified.
- In addition, there are specifications available to describe conditions at user-chosen points within the program body.

Tier One of the two levels of Larch/Ada specification. The interface tier is used to specify program behavior mathematically. The mathematical machinery for the specification is modularized into traits in the mathematical tier.

Trait A Larch theory module. Traits encapsulate all of the apparatus of a logical theory: the sorts, function declarations, axiom, lemmas, and so on. An example is the trait for the theory of integer arithmetic given on p. 83 of *Larch in Five Easy Pieces* [10]: the sort introduced is `Int`, a familiar axiom is $x < (x + 1)$.

Note that traits can be built up from other traits, just as program modules can call other program modules.

Predicate Transformation A process akin to symbolic execution of a program during which the precondition of each statement in a program is calculated.

VC See Verification Condition.

Verification Condition A formula that must be proved true for a program to be proved correct. Loosely speaking, the verification conditions of a program collectively assert that the statements of the program act together to transform the specified initial program state into one of the specified acceptable final states.

Verification conditions are calculated from specifications and preconditions.

Appendix C: Package Calendar_Utilities

This appendix shows the full Ada text of the Ada specification and the body of the package Calendar_Utilities. It differs only by reformatting from the text that NASA sent ORA.

C.1 Package Specification of Calendar_Utilities

```
with Calendar;
package Calendar_Utilities is

    type Year        is new Calendar.Year_Number;
    type Month       is range 1 .. 12;
    type Day         is range 1 .. 31;
    type Hour        is range 0 .. 23;
    type Minute      is range 0 .. 59;
    type Second      is range 0 .. 59;
    type Millisecond is range 0 .. 999;

    type Time is
    record
        The_Year        : Year;
        The_Month       : Month;
        The_Day         : Day;
        The_Hour        : Hour;
        The_Minute      : Minute;
        The_Second      : Second;
        The_Millisecond : Millisecond;
    end record;

    type Interval is
    record
        Elapsed_Days         : Natural;
        Elapsed_Hours        : Hour;
        Elapsed_Minutes      : Minute;
        Elapsed_Seconds      : Second;
        Elapsed_Milliseconds : Millisecond;
    end record;

    type Year_Day is range 1 .. 366;

    type Month_Name is (January, February, March,
                        April,   May,      June,
```

```
                    July,     August,    September,
                    October,  November,  December);
   type Day_Name    is (Monday,   Tuesday,  Wednesday,
                    Thursday, Friday,   Saturday, Sunday);

   type Period is (AM, PM);

   type Time_Format is (Full,            -- 01:21:06:30 PM
                        Military);       -- 13:21:06:30
   type Date_Format is (Full,            -- FEBRUARY 27, 1955
                        Month_Day_Year); -- 02/27/55

   function Is_Leap_Year  (The_Year  : in Year)
      return Boolean;
   function Days_In       (The_Year  : in Year)
      return Year_Day;
   function Days_In       (The_Month : in Month;
                           The_Year  : in Year)
      return Day;
   function Month_Of      (The_Month : in Month)
      return Month_Name;
   function Month_Of      (The_Month : in Month_Name)
      return Month;
   function Day_Of        (The_Year  : in Year;
                           The_Day   : in Year_Day)
      return Day_Name;
   function Day_Of        (The_Time  : in Time)
      return Year_Day;
   function Time_Of       (The_Year  : in Year;
                           The_Day   : in Year_Day)
      return Time;
   function Period_Of     (The_Time  : in Time)
      return Period;
   function Time_Of       (The_Time  : in Time)
      return Calendar.Time;
   function Time_Of       (The_Time  : in Calendar.Time)
      return Time;
   function Time_Image_Of (The_Time  : in Time;
                           Time_Form : in Time_Format := Full)
      return String;
   function Date_Image_Of (The_Time  : in Time;
                           Date_Form : in Date_Format := Full)
      return String;
   function Value_Of      (The_Date  : in String;
```

```
                        The_Time  : in String;
                        Date_Form : in Date_Format := Full;
                        Time_Form : in Time_Format := Full)
        return Time;

    function Duration_Of (The_Interval : in Interval)
        return Duration;
    function Interval_Of (The_Duration : in Duration)
        return Interval;
    function Image_Of    (The_Interval : in Interval)
        return String;
    function Value_Of    (The_Interval : in String)
        return Interval;

    Lexical_Error : exception;

end Calendar_Utilities;
```

C.2 Package Body of Calendar_Utilities

```
with Integer_Utilities,
     Fixed_Point_Utilities,
     String_Utilities;
package body Calendar_Utilities is

    type Month_Day is array(Month) of Day;

    Century_Offset          : constant           := 1900;
    Days_Per_Year           : constant           := 365;
    Days_Per_Month          : constant Month_Day := (1  => 31,
                                                     2  => 28,
                                                     3  => 31,
                                                     4  => 30,
                                                     5  => 31,
                                                     6  => 30,
                                                     7  => 31,
                                                     8  => 31,
                                                     9  => 30,
                                                     10 => 31,
                                                     11 => 30,
                                                     12 => 31);
    First_Day               : constant Day_Name  := Tuesday;
    Seconds_Per_Minute      : constant           := 60;
    Seconds_Per_Hour        : constant           :=
```

```
                              60 * Seconds_Per_Minute;
   Seconds_Per_Day          : constant           :=
                              24 * Seconds_Per_Hour;
   Milliseconds_Per_Second : constant           := 1000;
   Noon                     : constant Hour      := 12;
   Time_Separator           : constant Character := ':';
   Date_Separator           : constant Character := '/';
   Blank                    : constant Character := ' ';
   Comma                    : constant Character := ',';
   Zero                     : constant Character := '0';

   package Natural_Utilities is new Integer_Utilities
                                      (Number => Natural);

   package Duration_Utilities is new Fixed_Point_Utilities
                                       (Number => Duration);

   function Image_Of (The_Number : in Natural) return String is
   begin
       if The_Number < 10 then
           return String_Utilities.Replaced
                    (The_Character      => Blank,
                     With_The_Character => Zero,
                     In_The_String      =>
                       Natural_Utilities.Image_Of(The_Number));
       else
           return String_Utilities.Stripped_Leading
                    (The_Character   => Blank,
                     From_The_String =>
                       Natural_Utilities.Image_Of(The_Number));
       end if;
   end Image_Of;

   function Is_Leap_Year (The_Year : in Year) return Boolean is
   begin
       if The_Year mod 100 = 0 then
           return (The_Year mod 400 = 0);
       else
           return (The_Year mod 4 = 0);
       end if;
   end Is_Leap_Year;

   function Days_In (The_Year : in Year) return Year_Day is
   begin
```

```
    if Is_Leap_Year(The_Year) then
        return (Days_Per_Year + 1);
    else
       return Days_Per_Year;
    end if;
end Days_In;

function Days_In (The_Month : in Month;
                  The_Year  : in Year) return Day is
begin
    if (The_Month = Month_Name'Pos(February) + 1) and then
       Is_Leap_Year(The_Year) then
        return (Days_Per_Month(Month_Name'Pos(February) + 1)
                + 1);
    else
        return Days_Per_Month(The_Month);
    end if;
end Days_In;

function Month_Of (The_Month : in Month) return Month_Name is
begin
    return Month_Name'Val(The_Month - 1);
end Month_Of;

function Month_Of (The_Month : in Month_Name) return Month is
begin
    return (Month_Name'Pos(The_Month) + 1);
end Month_Of;

function Day_Of (The_Year : in Year;
                 The_Day  : in Year_Day) return Day_Name is
    Result : Day_Name := First_Day;
    procedure Increment(The_Day : in out Day_Name;
                        Offset  : in     Natural := 1) is
      --This moves you through day names by offset places.
      --Instead of using MOD, it uses exception when you pass
      --Sunday
    begin
        The_Day := Day_Name'Val(Day_Name'Pos(The_Day)
                                + Offset);
    exception
        when Constraint_Error =>
            The_Day := Day_Name'Val(Day_Name'Pos(The_Day)
                                    + Offset - 7);
```

```
    end Increment;
begin
    for Index in (Year'First + 1) .. The_Year loop
        if Is_Leap_Year(The_Year) then
            Increment(Result, Offset => 2);
        else
            Increment(Result);
        end if;
    end loop;
      --this uses 365/366 REM 7 and goes through the years,
      --then does days
    Increment(Result,
              Offset => Natural(((The_Day mod 7) - 1)));
    return Result;
end Day_Of;

function Day_Of (The_Time : in Time) return Year_Day is
    Result : Natural := 0;
begin
    for Index in Month'First .. (The_Time.The_Month - 1) loop
        Result := Result +
                  Natural(Days_In(Index, The_Time.The_Year));
    end loop;
    return Year_Day(Result + Natural(The_Time.The_Day));
end Day_Of;

function Time_Of (The_Year   : in Year;
                  The_Day    : in Year_Day) return Time is
    Result : Year_Day := The_Day;
begin
    for Index in Month'First .. Month'Last loop
        if Result <= Year_Day(Days_In(Index, The_Year)) then
            return Time'(
                    The_Year        => The_Year,
                    The_Month       => Index,
                    The_Day         => Day(Result),
                    The_Hour        => Hour'First,
                    The_Minute      => Minute'First,
                    The_Second      => Second'First,
                    The_Millisecond => Millisecond'First);
        else
            Result :=
              Result - Year_Day(Days_In(Index, The_Year));
        end if;
```

```
        end loop;
        raise Lexical_Error;
    end Time_Of;

    function Period_Of (The_Time : in Time) return Period is
    begin
        if The_Time.The_Hour >= Noon then
            return PM;
        else
            return AM;
        end if;
    end Period_Of;

    function Time_Of (The_Time : in Time) return Calendar.Time is
    begin
        return Calendar.Time_Of
                 (Year    =>
                    Calendar.Year_Number(The_Time.The_Year),
                  Month   =>
                    Calendar.Month_Number(The_Time.The_Month),
                  Day     =>
                    Calendar.Day_Number(The_Time.The_Day),
                  Seconds =>
                    Calendar.Day_Duration
                      (Integer(The_Time.The_Hour) *
                         Seconds_Per_Hour) +
                       Calendar.Day_Duration
                         (Integer(The_Time.The_Minute) *
                           Seconds_Per_Minute) +
                       Calendar.Day_Duration(
                         The_Time.The_Second) +
                       Calendar.Day_Duration
                        (The_Time.The_Millisecond /
                         Milliseconds_Per_Second));
    end Time_Of;

    function Time_Of (The_Time : in Calendar.Time) return Time is
        Result         : Time;
        Total_Duration : Calendar.Day_Duration;
        Seconds        : Natural;
    begin
        Calendar.Split(The_Time,
                       Year    =>
                         Calendar.Year_Number(Result.The_Year),
```

```
                    Month   =>
                      Calendar.Month_Number(Result.The_Month),
                    Day     =>
                      Calendar.Day_Number(Result.The_Day),
                    Seconds =>
                      Total_Duration);
    Seconds := Duration_Utilities.Floor(Total_Duration);
    Result.The_Hour := Hour(Seconds / Seconds_Per_Hour);
    Seconds := Seconds mod Seconds_Per_Hour;
    Result.The_Minute :=
      Minute(Seconds / Seconds_Per_Minute);
    Result.The_Second :=
      Second(Seconds mod Seconds_Per_Minute);
    Result.The_Millisecond :=
      Millisecond
        (Duration_Utilities.Real_Part
          (Total_Duration) * Milliseconds_Per_Second);
    return Result;
end Time_Of;

function Time_Image_Of (The_Time  : in Time;
                        Time_Form : in Time_Format := Full)
return String is
begin
    case Time_Form is
        when Full     =>
            if The_Time.The_Hour > Noon then
                return (
                    Image_Of(Natural(
                      The_Time.The_Hour - 12)) &
                    Time_Separator &
                    Image_Of(Natural(The_Time.The_Minute)) &
                    Time_Separator &
                    Image_Of(Natural(The_Time.The_Second)) &
                    Time_Separator &
                    Image_Of(Natural(
                      The_Time.The_Millisecond) / 10) &
                    " PM");
            else
                return (
                  Image_Of(Natural(The_Time.The_Hour)) &
                  Time_Separator &
                  Image_Of(Natural(The_Time.The_Minute)) &
                  Time_Separator &
```

```
                    Image_Of(Natural(The_Time.The_Second)) &
                    Time_Separator &
                    Image_Of(Natural(
                      The_Time.The_Millisecond) / 10) &
                    " AM");
            end if;
        when Military =>
            return (Image_Of(Natural(The_Time.The_Hour)) &
                    Time_Separator &
                    Image_Of(Natural(The_Time.The_Minute)) &
                    Time_Separator &
                    Image_Of(Natural(The_Time.The_Second)) &
                    Time_Separator &
                    Image_Of(Natural(
                      The_Time.The_Millisecond) / 10));
    end case;
end Time_Image_Of;

function Date_Image_Of (The_Time  : in Time;
                        Date_Form : in Date_Format := Full)
return String is
begin
    case Date_Form is
        when Full           =>
            return (Month_Name'Image(Month_Name'Val(
                      The_Time.The_Month - 1)) &
                    Natural_Utilities.Image_Of(
                      Natural(The_Time.The_Day)) &
                    Comma &
                    Natural_Utilities.Image_Of
                      (Natural(The_Time.The_Year)));
        when Month_Day_Year =>
            return (Image_Of(Integer(The_Time.The_Month)) &
                    Date_Separator &
                    Image_Of(Integer(The_Time.The_Day)) &
                    Date_Separator &
                    Image_Of(Integer(The_Time.The_Year))
                      (4 .. 5));
    end case;
end Date_Image_Of;

function Value_Of (The_Date  : in String;
                   The_Time  : in String;
                   Date_Form : in Date_Format := Full;
```

```
                    Time_Form : in Time_Format := Full)
    return Time is
        Result      : Time;
        Left_Index  : Positive;
        Right_Index : Positive;
    begin
        case Date_Form is
           when Full           =>
               Right_Index := String_Utilities.Location_Of
                                (The_Character => Blank,
                                 In_The_String => The_Date);
               Result.The_Month := Month(Month_Name'Pos
                                     (Month_Name'Value
                                        (The_Date(
                                           The_Date'First ..
                                           (Right_Index - 1))))
                                      + 1);
               Left_Index := Right_Index + 1;
               Right_Index := String_Utilities.Location_Of
                                (The_Character => Comma,
                                 In_The_String =>
                                   The_Date(
                                     Left_Index..The_Date'Last));
               Result.The_Day := Day(Natural_Utilities.Value_Of
                                     (The_Date
                                       (Left_Index ..
                                       (Right_Index - 1))));
               Left_Index := Right_Index + 1;
               Result.The_Year := Year(Natural_Utilities.Value_Of
                                      (The_Date
                                        (Left_Index ..
                                         The_Date'Last)));
           when Month_Day_Year =>
               Right_Index := String_Utilities.Location_Of
                                (The_Character => Date_Separator,
                                 In_The_String => The_Date);
               Result.The_Month :=
                 Month(Natural_Utilities.Value_Of
                   (The_Date
                     (The_Date'First .. (Right_Index - 1))));
               Left_Index := Right_Index + 1;
               Right_Index := String_Utilities.Location_Of
                                (The_Character => Date_Separator,
                                 In_The_String =>
```

```
                              The_Date(Left_Index ..
                                       The_Date'Last));
          Result.The_Day := Day(Natural_Utilities.Value_Of
                              (The_Date
                                (Left_Index ..
                                (Right_Index - 1))));
          Left_Index := Right_Index + 1;
          Result.The_Year := Year(Natural_Utilities.Value_Of
                                    (The_Date
                                      (Left_Index ..
                                      The_Date'Last)) +
                                  Natural(Century_Offset));
   end case;
   Right_Index := String_Utilities.Location_Of
                    (The_Character => Time_Separator,
                     In_The_String => The_Time);
   Result.The_Hour := Hour(Natural_Utilities.Value_Of
                         (The_Time
                           (The_Time'First ..
                           (Right_Index - 1))));
   Left_Index := Right_Index + 1;
   Right_Index := String_Utilities.Location_Of
                    (The_Character => Time_Separator,
                     In_The_String =>
                       The_Time(Left_Index..The_Time'Last));
   Result.The_Minute := Minute(Natural_Utilities.Value_Of
                           (The_Time
                             (Left_Index ..
                             (Right_Index - 1))));
   Left_Index := Right_Index + 1;
   Right_Index := String_Utilities.Location_Of
                    (The_Character => Time_Separator,
                     In_The_String =>
                       The_Time(Left_Index..The_Time'Last));
   Result.The_Second := Second(Natural_Utilities.Value_Of
                           (The_Time
                             (Left_Index ..
                             (Right_Index - 1))));
   Left_Index := Right_Index + 1;
   case Time_Form is
      when Full     =>
          Right_Index := String_Utilities.Location_Of
                           (The_Character => Blank,
                            In_The_String =>
```

```
                                The_Time(Left_Index ..
                                The_Time'Last));
            Result.The_Millisecond :=
             Millisecond(Natural_Utilities.Value_Of
                (The_Time (Left_Index .. (Right_Index - 1))));
            Left_Index := Right_Index + 1;
            if Period'Value(The_Time(Left_Index ..
                            The_Time'Last)) = PM
            then
                if Result.The_Hour /= Noon then
                    Result.The_Hour :=
                      Result.The_Hour + Noon;
                end if;
            end if;
        when Military =>
            Result.The_Millisecond :=
              Millisecond(Natural_Utilities.Value_Of
                (The_Time (Left_Index .. The_Time'Last)));
    end case;
    return Result;
exception
    when Constraint_Error =>
        raise Lexical_Error;
end Value_Of;

function Duration_Of (The_Interval : in Interval)
return Duration is
begin
    return (Duration(Integer(The_Interval.Elapsed_Days)
            * Seconds_Per_Day) +
            Duration(Integer(The_Interval.Elapsed_Hours)
            * Seconds_Per_Hour)
+
            Duration(Integer(The_Interval.Elapsed_Minutes)
            * Seconds_Per_Minute) +
            Duration(The_Interval.Elapsed_Seconds) +
            Duration(The_Interval.Elapsed_Milliseconds
                    / 1000));
end Duration_Of;

function Interval_Of (The_Duration : in Duration)
return Interval is
    Result      : Interval;
    The_Seconds : Duration := The_Duration;
```

```
begin
    Result.Elapsed_Days := Duration_Utilities.Floor
                              (The_Seconds / Seconds_Per_Day);
    The_Seconds :=
      The_Seconds -
        Duration(Integer(Result.Elapsed_Days)
        * Seconds_Per_Day);
    Result.Elapsed_Hours :=
      Hour(Duration_Utilities.Floor
        (The_Seconds / Seconds_Per_Hour));
    The_Seconds :=
      The_Seconds -
        Duration(Integer(Result.Elapsed_Hours)
        * Seconds_Per_Hour);
    Result.Elapsed_Minutes :=
      Minute(Duration_Utilities.Floor
        (The_Seconds / Seconds_Per_Minute));
    The_Seconds :=
      The_Seconds -
        Duration(Integer(Result.Elapsed_Minutes)
        * Seconds_Per_Minute);
    Result.Elapsed_Seconds :=
      Second(Duration_Utilities.Floor
                                (The_Seconds));
    The_Seconds :=
      The_Seconds - Duration(Result.Elapsed_Seconds);
    Result.Elapsed_Milliseconds :=
      Millisecond
        (Duration_Utilities.Floor(The_Seconds * 1000));
    return Result;
end Interval_Of;

function Image_Of (The_Interval : in Interval)
return String is
begin
    return (Image_Of(Natural(The_Interval.Elapsed_Days)) &
            Time_Separator &
            Image_Of(Natural(The_Interval.Elapsed_Hours)) &
            Time_Separator &
            Image_Of(Natural(The_Interval.Elapsed_Minutes)) &
            Time_Separator &
            Image_Of(Natural(The_Interval.Elapsed_Seconds)) &
            Time_Separator &
            Image_Of(Natural(
```

```
                The_Interval.Elapsed_Milliseconds)));
    end Image_Of;

    function Value_Of (The_Interval : in String)
    return Interval is
        Result      : Interval;
        Left_Index  : Positive;
        Right_Index : Positive;
    begin
        Right_Index := String_Utilities.Location_Of
                        (The_Character => Time_Separator,
                         In_The_String => The_Interval);
        Result.Elapsed_Days := Natural_Utilities.Value_Of
                                (The_Interval
                                  (The_Interval'First ..
                                  (Right_Index - 1)));
        Left_Index := Right_Index + 1;
        Right_Index := String_Utilities.Location_Of
                       (The_Character => Time_Separator,
                        In_The_String =>
                          The_Interval(Left_Index ..
                          The_Interval'Last));
        Result.Elapsed_Hours := Hour(Natural_Utilities.Value_Of
                                  (The_Interval
                                    (Left_Index ..
                                    (Right_Index - 1))));
        Left_Index := Right_Index + 1;
        Right_Index := String_Utilities.Location_Of
                       (The_Character => Time_Separator,
                        In_The_String =>
                          The_Interval(Left_Index ..
                                       The_Interval'Last));
        Result.Elapsed_Minutes :=
           Minute(Natural_Utilities.Value_Of
             (The_Interval
               (Left_Index .. (Right_Index - 1))));
        Left_Index := Right_Index + 1;
        Right_Index := String_Utilities.Location_Of
                       (The_Character => Time_Separator,
                        In_The_String =>
                          The_Interval(Left_Index ..
                                       The_Interval'Last));
        Result.Elapsed_Seconds :=
          Second(Natural_Utilities.Value_Of
```

```
            (The_Interval
              (Left_Index .. (Right_Index - 1))));
        Left_Index := Right_Index + 1;
        Result.Elapsed_Milliseconds :=
          Millisecond
            (Natural_Utilities.Value_Of
              (The_Interval
                (Left_Index .. The_Interval'Last)));
        return Result;
    exception
        when Constraint_Error =>
            raise Lexical_Error;
    end Value_Of;

end Calendar_Utilities;
```

Appendix D: Package Calendar

This appendix shows the **Calendar** package of the Ada Reference Manual.


```
package calendar is
  type time is private;

  subtype year_number  is integer  range 1901 .. 2099;
  subtype month_number is integer  range 1 .. 12;
  subtype day_number   is integer  range 1 .. 31;
  subtype day_duration is duration range 0.0 .. 86_400.0;

  function clock return time;

  function year    (date : time) return year_number;
  function month   (date : time) return month_number;
  function day     (date : time) return day_number;
  function seconds (date : time) return day_duration;

  procedure split (date    : in  time;
                   year    : out year_number;
                   month   : out month_number;
                   day     : out day_number;
                   seconds : out day_duration);

  function time_of(year    : year_number;
                   month   : month_number;
                   day     : day_number;
                   seconds : day_duration := 0.0) return time;

  function "+" (left : time;     right: duration) return time;
  function "+" (left : duration; right: time )    return time;
  function "-" (left : time;     right: duration) return time;
  function "-" (left : time;     right: time )
        return duration;

  function "<"  (left, right : time) return boolean;
  function "<=" (left, right : time) return boolean;
  function ">"  (left, right : time) return boolean;
  function ">=" (left, right : time) return boolean;

  time error : exception;
```

```
        -- can be raised by time_of, "+" and "-"

private
  -- implementation-dependent
end;
```

Appendix E: Full Output from the Penelope Verification


```
--! Verification status: Not verified
      (21 VCs generated; 2 VCs hidden; 16 VCs proved)
--| TRAIT Z IS
--| DECLARES
--| INTRODUCES
--| Int_identity: Int -> Int;
--| AXIOMS: FORALL  [n:Int]
--| Int_identity: (Int_identity(n)=n);
--| END AXIOMS;
--| LEMMAS: FORALL  [a:Int, m:Int, m1:Int, m2:Int]
--| mod_lower: ((m>0)->(0<=(a MOD m)));
--| mod_upper: ((m>0)->((a MOD m)<m));
--| mod_ident: (((0<=a) AND (a<m))->((a MOD m)=a));
--| mod_transitive: (
    (
      (((m1/=0) AND (m2/=0)) AND ((m1 MOD m2)=0))
     AND
      ((a MOD m1)=0))
  ->
    ((a MOD m2)=0));
--| mod_subtract: (
    (((m>0) AND (m<=a)) AND (a<=((2*m)-1)))
  ->
    ((a MOD m)=(a-m)));
--| div_upper: (((0<=a) AND (a<m))->((a/m)=0));
--| END LEMMAS;
--| TRAIT fixed_point IS
--| DECLARES
--| SORT fixnum IS AnyRecordSort;
--| INTRODUCES
--| good_fixnum: fixnum -> Bool;
--| abstract: fixnum -> Rational;
--| pair: Int, Int -> fixnum;
--| Int_to_fixnum: Int -> fixnum;
--| "*": fixnum, fixnum -> fixnum;
--| "*": fixnum, Int -> fixnum;
--| "*": Int, fixnum -> fixnum;
```

```
--| "/": fixnum, fixnum -> fixnum;
--| "/": fixnum, Int -> fixnum;
--| "+": fixnum, fixnum -> fixnum;
--| "-": fixnum, fixnum -> fixnum;
--| "<": fixnum, fixnum -> Bool;
--| "<=": fixnum, fixnum -> Bool;
--| ">": fixnum, fixnum -> Bool;
--| ">=": fixnum, fixnum -> Bool;
--| NonZero: fixnum -> Bool;
--| IntegerPart: fixnum -> Int;
--| RealPart: fixnum -> fixnum;
--| Floor: fixnum -> Int;
--| Ceiling: fixnum -> Int;
--| fixnum_abs: fixnum -> fixnum;
--| AXIOMS: FORALL  [m:fixnum, n:fixnum, i:Int, j:Int]
--| good_fixnum: (good_fixnum(m)=(m.denom>0));
--| concrete_equality: (
    ((m.numer=n.numer) AND (m.denom=n.denom))
  ->
    (m=n));
--| abstract_equality: (
    (
      (good_fixnum(m) AND good_fixnum(n))
     AND
      ((m.numer*n.denom)=Int_identity((m.denom*n.numer))))
  ->
    (abstract(m)=abstract(n)));
--| pair_numer: (pair(i, j).numer=i);
--| pair_denom: (pair(i, j).denom=j);
--| Int_to_fixnum: (Int_to_fixnum(i)=pair(i, 1));
--| times_ff_numer: (
    (good_fixnum(m) AND good_fixnum(n))
  ->
    ((m*n).numer=Int_identity((m.numer*n.numer))));
--| times_ff_denom: (
    (good_fixnum(m) AND good_fixnum(n))
  ->
    ((m*n).denom=Int_identity((m.denom*n.denom))));
--| times_fi_numer: (
    good_fixnum(m)
  ->
    ((m*i).numer=Int_identity((m.numer*i))));
--| times_fi_denom: (good_fixnum(m)->((m*i).denom=m.denom));
--| times_if_numer: (
```

```
    good_fixnum(m)
  ->
    ((i*m).numer=Int_identity((i*m.numer))));
--| times_if_denom: (good_fixnum(m)->((i*m).denom=m.denom));
--| divides_ff_numer: (
    ((good_fixnum(m) AND good_fixnum(n)) AND NonZero(n))
  ->
    ((m/n).numer=Int_identity((m.numer*n.denom))));
--| divides_ff_denom: (
    ((good_fixnum(m) AND good_fixnum(n)) AND NonZero(n))
  ->
    ((m/n).denom=Int_identity((m.denom*n.numer))));
--| divides_fi_numer: (
    (good_fixnum(m) AND (i/=0))
  ->
    ((m/i).numer=m.numer));
--| divides_fi_denom: (
    (good_fixnum(m) AND (i/=0))
  ->
    ((m/i).denom=Int_identity((m.denom*i))));
--| plus_ff_numer: (
    (good_fixnum(m) AND good_fixnum(n))
  ->
    (
      (m+n).numer
    =
      Int_identity(((m.numer*n.denom)+(n.numer*m.denom)))));
--| plus_ff_denom: (
    (good_fixnum(m) AND good_fixnum(n))
  ->
    ((m+n).denom=Int_identity((m.denom*n.denom))));
--| minus_ff_numer: (
    (good_fixnum(m) AND good_fixnum(n))
  ->
    (
      (m-n).numer
    =
      Int_identity(((m.numer*n.denom)-(n.numer*m.denom)))));
--| minus_ff_denom: (
    (good_fixnum(m) AND good_fixnum(n))
  ->
    ((m-n).denom=Int_identity((m.denom*n.denom))));
--| fixnum_lt: (
    (good_fixnum(m) AND good_fixnum(n))
```

```
  ->
    (
      (m<n)
    =
      (
        Int_identity((m.numer*n.denom))
      <
        Int_identity((n.numer*m.denom)))));
--| fixnum_le: (
    (good_fixnum(m) AND good_fixnum(n))
  ->
    (
      (m<=n)
    =
      (
        Int_identity((m.numer*n.denom))
      <=
        Int_identity((n.numer*m.denom)))));
--| fixnum_gt: (
    (good_fixnum(m) AND good_fixnum(n))
  ->
    (
      (m>n)
    =
      (
        Int_identity((m.numer*n.denom))
      >
        Int_identity((n.numer*m.denom)))));
--| fixnum_ge: (
    (good_fixnum(m) AND good_fixnum(n))
  ->
    (
      (m>=n)
    =
      (
        Int_identity((m.numer*n.denom))
      >=
        Int_identity((n.numer*m.denom)))));
--| IntegerPart: (
    good_fixnum(m)
  ->
    (IntegerPart(n)=Int_identity((n.numer/n.denom))));
--| RealPart: (
    good_fixnum(m)
```

```
  ->
    (RealPart(m)=fixnum_abs((m-Int_to_fixnum(IntegerPart(n)))));
--| Floor: (
    good_fixnum(m)
  ->
    (
      Floor(m)
    =
      Int_identity(
        (
        IF ((m.numer<0) AND ((m.numer MOD m.denom)/=0))
          THEN ((m.numer/m.denom)-1)
           ELSE (m.numer/m.denom)));
--| fixnum_abs: (
    good_fixnum(m)
  ->
    (
      fixnum_abs(m)
    =
      (IF (m<Int_to_fixnum(0)) THEN ((-1)*m)  ELSE m)));
--| END AXIOMS;
--| LEMMAS: FORALL  [m:fixnum, n:fixnum, p:fixnum, i:Int,
n1:Int, d1:Int, n2:Int, d2:Int]
--| good_fixnum_real_part: (
    good_fixnum(m)
  ->
    good_fixnum(RealPart(m)));
--| concrete_equality_pair: (FORALL n1:Int, d1:Int, n2:Int,
  d2:Int::(((n1*d2)=(n2*d1))->(pair(n1, d1)=pair(n2, d2))));
--| floor1: ((Int_to_fixnum(0)<=m)->(0<=Floor(m)));
--| pair_plus: (
    (pair(n1, d1)+pair(n2, d2))
  =
    pair((Int_identity((n1*d2))+Int_identity((n2*d1))),
      (d1*d2)));
--| pair_minus: (
    (pair(n1, d1)-pair(n2, d2))
  =
    pair((Int_identity((n1*d2))-Int_identity((n2*d1))),
      (d1*d2)));
--| divides_pi: (FORALL n:Int, d:Int,
  i:Int::(((d/=0) AND (i/=0))->((pair(n, d)/i)=pair(n, (d*i)))));
--| END LEMMAS;
--| TRAIT sort_names IS
```

```
--| DECLARES
--| SORT calendar_time_sort IS Int;
--| SORT century IS (twentieth, twentyfirst);
--| SORT years IS Int;
--| SORT months IS Int;
--| SORT year_days IS Int;
--| SORT month_days IS Int;
--| SORT absolute_days IS Int;
--| SORT hours IS Int;
--| SORT minutes IS Int;
--| SORT seconds IS Int;
--| SORT milliseconds IS Int;
--| SORT time_sort IS AnyRecordSort;
--| SORT duration_sort IS fixnum;
--| SORT month_name_sort IS (January, February, March, April,
May, June, July, August, September, October, November, December);
--| SORT day_name_sort IS (Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday);
--| SORT period_sort IS (am, pm);
--| SORT time_format_sort IS (full, military);
--| SORT date_format_sort IS (full, month_day_year);
--| SORT interval_sort IS AnyRecordSort;
--| SORT catu IS Int;
--| INTRODUCES
--| month_name_sort_tic_pos: month_name_sort -> Int;
--| month_name_sort_tic_val: Int -> month_name_sort;
--| good_day_name: day_name_sort -> Bool;
--| day_name_sort_tic_pos: day_name_sort -> Int;
--| day_name_sort_tic_val: Int -> day_name_sort;
--| day_name_sort_increment: day_name_sort, Int -> day_name_sort;
--| milliseconds_tic_first: -> milliseconds;
--| seconds_tic_first: -> seconds;
--| minutes_tic_first: -> minutes;
--| hours_tic_first: -> hours;
--| years_tic_first: -> years;
--| months_tic_first: -> months;
--| months_tic_last: -> months;
--| month_name_sort_tic_first: -> month_name_sort;
--| month_name_sort_tic_last: -> month_name_sort;
--| time_sort_tic: -> time_sort;
--| month_days_tic: -> month_days;
--| extract_year: calendar_time_sort -> years;
--| extract_month: calendar_time_sort -> months;
--| extract_day: calendar_time_sort -> month_days;
```

```
--| extract_duration: calendar_time_sort -> duration_sort;
--| seconds_to_duration: seconds -> duration_sort;
--| "+": calendar_time_sort, duration_sort -> calendar_time_sort;
--| "-": calendar_time_sort, duration_sort -> calendar_time_sort;
--| calendar_time_sort_identity:
      calendar_time_sort -> calendar_time_sort;
--| duration_sort_identity: duration_sort -> duration_sort;
--| good_duration: duration_sort -> Bool;
--| good_day: absolute_days -> Bool;
--| good_hour: hours -> Bool;
--| good_minute: minutes -> Bool;
--| good_second: seconds -> Bool;
--| good_millisecond: milliseconds -> Bool;
--| good_interval: interval_sort -> Bool;
--| AXIOMS: FORALL  [the_duration:duration_sort, n:Int,
the_day_name:day_name_sort, offset:Int,
the_interval:interval_sort, the_day:absolute_days,
the_hour:hours, the_minute:minutes, the_second:seconds,
the_millisecond:milliseconds]
--| months_tic_first: (months_tic_first=1);
--| months_tic_last: (months_tic_last=12);
--| month_pos_january: (month_name_sort_tic_pos(January)=0);
--| month_pos_february: (month_name_sort_tic_pos(February)=1);
--| month_pos_march: (month_name_sort_tic_pos(March)=2);
--| month_pos_april: (month_name_sort_tic_pos(April)=3);
--| month_pos_may: (month_name_sort_tic_pos(May)=4);
--| month_pos_june: (month_name_sort_tic_pos(June)=5);
--| month_pos_july: (month_name_sort_tic_pos(July)=6);
--| month_pos_august: (month_name_sort_tic_pos(August)=7);
--| month_pos_september: (month_name_sort_tic_pos(September)=8);
--| month_pos_october: (month_name_sort_tic_pos(October)=9);
--| month_pos_november: (month_name_sort_tic_pos(November)=10);
--| month_pos_december: (month_name_sort_tic_pos(December)=11);
--| month_val_0: (month_name_sort_tic_val(0)=January);
--| month_val_1: (month_name_sort_tic_val(1)=February);
--| month_val_2: (month_name_sort_tic_val(2)=March);
--| month_val_3: (month_name_sort_tic_val(3)=April);
--| month_val_4: (month_name_sort_tic_val(4)=May);
--| month_val_5: (month_name_sort_tic_val(5)=June);
--| month_val_6: (month_name_sort_tic_val(6)=July);
--| month_val_7: (month_name_sort_tic_val(7)=August);
--| month_val_8: (month_name_sort_tic_val(8)=September);
--| month_val_9: (month_name_sort_tic_val(9)=October);
--| month_val_10: (month_name_sort_tic_val(10)=November);
```

```
--| month_val_11: (month_name_sort_tic_val(11)=December);
--| good_day_name: (
    good_day_name(the_day_name)
  =
    ((Monday<=the_day_name) OR (the_day_name<=Sunday)));
--| day_val_0: (day_name_sort_tic_val(0)=Monday);
--| day_val_1: (day_name_sort_tic_val(1)=Tuesday);
--| day_val_2: (day_name_sort_tic_val(2)=Wednesday);
--| day_val_3: (day_name_sort_tic_val(3)=Thursday);
--| day_val_4: (day_name_sort_tic_val(4)=Friday);
--| day_val_5: (day_name_sort_tic_val(5)=Saturday);
--| day_val_6: (day_name_sort_tic_val(6)=Sunday);
--| first_millisecond: (milliseconds_tic_first=0);
--| first_second: (seconds_tic_first=0);
--| first_minute: (minutes_tic_first=0);
--| first_hour: (hours_tic_first=0);
--| first_year: (years_tic_first=1901);
--| first_month: (month_name_sort_tic_first=January);
--| last_month: (month_name_sort_tic_last=December);
--| duration_sort_identity: (
    duration_sort_identity(the_duration)
  =
    the_duration);
--| day_name_sort_increment: (
    day_name_sort_increment(the_day_name, offset)
  =
    day_name_sort_tic_val(
      ((day_name_sort_tic_pos(the_day_name)+offset) MOD 7)));
--| good_duration: (
    good_duration(the_duration)
  =
    good_fixnum(the_duration));
--| good_day: (good_day(the_day)=(0<=the_day));
--| good_hour: (
    good_hour(the_hour)
  =
    ((0<=the_hour) AND (the_hour<24)));
--| good_minute: (
    good_minute(the_minute)
  =
    ((0<=the_minute) AND (the_minute<60)));
--| good_second: (
    good_second(the_second)
  =
```

```
    ((0<=the_second) AND (the_second<60)));
--| good_millisecond: (
    good_millisecond(the_millisecond)
  =
    ((0<=the_millisecond) AND (the_millisecond<1000)));
--| good_interval: (
    good_interval(the_interval)
  =
    (
      (
        (
          (
            good_day(the_interval.elapsed_days)
           AND
            good_hour(the_interval.elapsed_hours))
         AND
          good_minute(the_interval.elapsed_minutes))
       AND
        good_second(the_interval.elapsed_seconds))
     AND
      good_millisecond(the_interval.elapsed_milliseconds)));
--| END AXIOMS;
--| LEMMAS: FORALL  [the_day_name:day_name_sort, the_pos:Int,
the_second:seconds]
--| day_name_sort_tic_pos: (
    (
      (
        (
          (
            (
              (day_name_sort_tic_pos(Monday)=0)
             AND
              (day_name_sort_tic_pos(Tuesday)=1))
           AND
            (day_name_sort_tic_pos(Wednesday)=2))
         AND
          (day_name_sort_tic_pos(Thursday)=3))
       AND
        (day_name_sort_tic_pos(Friday)=4))
     AND
      (day_name_sort_tic_pos(Saturday)=5))
   AND
    (day_name_sort_tic_pos(Sunday)=6));
--| day_name_sort_tic_pos_range: (
```

```
    (0<=day_name_sort_tic_pos(the_day_name))
   AND
    (day_name_sort_tic_pos(the_day_name)<=6));
--| day_name_sort_tic_val_range: (
    ((0<=the_pos) AND (the_pos<=6))
  ->
    (
      (Monday<=day_name_sort_tic_val(the_pos))
     AND
      (day_name_sort_tic_val(the_pos)<=Sunday)));
--| seconds_to_duration: (
    seconds_to_duration(the_second)
  =
    pair(the_second, 1));
--| END LEMMAS;
--| TRAIT conversion_factors IS
--| DECLARES
--| INTRODUCES
--| days_to_hours: -> hours;
--| hours_to_minutes: -> minutes;
--| hours_to_seconds: -> seconds;
--| days_to_seconds: -> seconds;
--| minutes_to_seconds: -> seconds;
--| days_to_milliseconds: -> milliseconds;
--| seconds_to_milliseconds: -> milliseconds;
--| seconds_to_catu: -> catu;
--| milliseconds_to_catu: -> catu;
--| AXIOMS:
--| hours_to_minutes: (hours_to_minutes=60);
--| days_to_hours: (days_to_hours=24);
--| minutes_to_seconds: (minutes_to_seconds=60);
--| seconds_to_milliseconds: (seconds_to_milliseconds=1000);
--| days_to_seconds: (
    days_to_seconds
  =
    ((days_to_hours*hours_to_minutes)*minutes_to_seconds));
--| hours_to_seconds: (
    hours_to_seconds
  =
    (hours_to_minutes*minutes_to_seconds));
--| milliseconds_to_catu: (milliseconds_to_catu>0);
--| seconds_to_catu: (
    seconds_to_catu
  =
```

```
    (seconds_to_milliseconds*milliseconds_to_catu));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;
--| TRAIT time_representations IS
--| DECLARES
--| INTRODUCES
--| days_in_month: months, years -> month_days;
--| is_leap_year: years -> Bool;
--| is_non_leap_centennial: years -> Bool;
--| days_in_year: years -> year_days;
--| days_of_years_since_1900: years -> absolute_days;
--| seconds_since_1900: time_sort -> seconds;
--| days_since_Jan: months, years -> year_days;
--| day_name_of: years, year_days -> day_name_sort;
--| day_of_year: years, months, month_days -> year_days;
--| month_of: years, year_days -> months;
--| day_of_month: years, year_days -> month_days;
--| year_of: absolute_days -> years;
--| good_year: years -> Bool;
--| good_month: months -> Bool;
--| good_month_day: month_days -> Bool;
--| good_date: years, months, month_days, duration_sort -> Bool;
--| good_time: time_sort -> Bool;
--| good_year_and_day: years, year_days -> Bool;
--| day_name_sort_val: Int -> day_name_sort;
--| time_of: years,
  months,
  month_days,
  duration_sort -> calendar_time_sort;
--| AXIOMS: FORALL  [the_month:Int, the_year:Int,
the_time:time_sort, the_year_day:Int, the_month_day:Int,
the_absolute_day:Int]
--| days_in_January: (days_in_month(1, the_year)=31);
--| days_in_February: (
    days_in_month(2, the_year)
  =
    (IF is_leap_year(the_year) THEN 29  ELSE 28));
--| days_in_March: (days_in_month(3, the_year)=31);
--| days_in_April: (days_in_month(4, the_year)=30);
--| days_in_May: (days_in_month(5, the_year)=31);
--| days_in_June: (days_in_month(6, the_year)=30);
--| days_in_July: (days_in_month(7, the_year)=31);
--| days_in_August: (days_in_month(8, the_year)=31);
```

```
--| days_in_September: (days_in_month(9, the_year)=30);
--| days_in_October: (days_in_month(10, the_year)=31);
--| days_in_November: (days_in_month(11, the_year)=30);
--| days_in_December: (days_in_month(12, the_year)=31);
--| leap_year: (
    is_leap_year(the_year)
  =
    (
      ((the_year MOD 4)=0)
     AND
      (NOT is_non_leap_centennial(the_year))));
--| non_leap_centennial: (
    is_non_leap_centennial(the_year)
  =
    (((the_year MOD 100)=0) AND ((the_year MOD 400)/=0)));
--| days_in_year: (
    days_in_year(the_year)
  =
    (IF is_leap_year(the_year) THEN 366  ELSE 365));
--| days_since_1900_0: (days_of_years_since_1900(1900)=0);
--| days_since_1900_1: (
    (the_year>1900)
  ->
    (
      days_of_years_since_1900(the_year)
    =
      (
        days_in_year((the_year-1))
      +
        days_of_years_since_1900((the_year-1)))));
--| days_since_January_0: (days_since_Jan(1, the_year)=0);
--| days_since_January_1: (
    (the_month>1)
  ->
    (
      days_since_Jan(the_month, the_year)
    =
      (
        days_in_month((the_month-1), the_year)
      +
        days_since_Jan((the_month-1), the_year))));
--| good_year: (
    good_year(the_year)
  =
```

```
    ((1901<=the_year) AND (the_year<=2099)));
--| good_month: (
    good_month(the_month)
  =
    ((1<=the_month) AND (the_month<=12)));
--| good_month_day: (
    good_month_day(the_month_day)
  =
    ((1<=the_month_day) AND (the_month_day<=31)));
--| good_time: (
    good_time(the_time)
  =
    (
      (
        (
          (
            (
              (
                (
                  (
                    (
                      (
                        (
                          (
                            (1901<=the_time.the_year)
                           AND
                            (the_time.the_year<=2099))
                          AND
                           (1<=the_time.the_month))
                         AND
                          (the_time.the_month<=12))
                        AND
                         (1<=the_time.the_day))
                       AND
                        (
                          the_time.the_day
                         <=
                          days_in_month(the_time.the_month,
                            the_time.the_year)))
                     AND
                      (0<=the_time.the_hour))
                   AND
                    (the_time.the_hour<days_to_hours))
```

```
              AND
               (0<=the_time.the_minute))
             AND
              (the_time.the_minute<hours_to_minutes))
            AND
             (0<=the_time.the_second))
           AND
            (the_time.the_second<minutes_to_seconds))
          AND
           (0<=the_time.the_millisecond))
         AND
          (the_time.the_millisecond<seconds_to_milliseconds)));
--| good_year_and_day: (
    good_year_and_day(the_year, the_year_day)
  =
    (
      ((1901<=the_year) AND (the_year<=2099))
     AND
      (
        (1<=the_year_day)
       AND
        (the_year_day<=days_in_year(the_year)))));
--| day_name: (
    day_name_of(the_year, the_year_day)
  =
    day_name_sort_tic_val(
      (
        (days_of_years_since_1900(the_year)+(the_year_day-1))
       MOD
        7)));
--| day_of_year: (
    day_of_year(the_year, the_month, the_month_day)
  =
    (days_since_Jan(the_month, the_year)+the_month_day));
--| month_of: (
    (
      (days_since_Jan(the_month, the_year)<the_year_day)
     AND
      (
        the_year_day
       <=
        (
          days_since_Jan(the_month, the_year)
         +
```

```
          days_in_month(the_month, the_year))))
  ->
    (month_of(the_year, the_year_day)=the_month));
--| day_of_month: (
    day_of_month(the_year, the_year_day)
  =
    (
      the_year_day
    -
      days_since_Jan(month_of(the_year, the_year_day),
        the_year)));
--| year_of: (
    (
      (days_of_years_since_1900(the_year)<the_absolute_day)
     AND
      (the_absolute_day<=days_of_years_since_1900((the_year+1))))
  ->
    (year_of(the_absolute_day)=the_year));
--| seconds_since_1900: (
    seconds_since_1900(the_time)
  =
    (
      (
        (
          (
            (
              days_of_years_since_1900(the_time.the_year)
            *
              days_to_seconds)
          +
            Int_identity(
              (
                days_since_Jan(the_time.the_month,
                  the_time.the_year)
              *
                days_to_seconds)))
        +
          Int_identity((the_time.the_hour*hours_to_seconds)))
      +
        Int_identity((the_time.the_minute*minutes_to_seconds)))
    +
      duration_sort_identity(the_time.the_second)));
--| END AXIOMS;
--| LEMMAS: FORALL  [the_year:years, the_month:months]
```

```
--| day_name_of: (day_name_of(1900, 1)=Tuesday);
--| days_in_months: (
    days_in_month(the_month, the_year)
  =
    (
    IF (the_month=1)
      THEN 31
       ELSE (
        IF (the_month=2)
          THEN (IF is_leap_year(the_year) THEN 29  ELSE 28)
           ELSE (
            IF (the_month=3)
              THEN 31
               ELSE (
                IF (the_month=4)
                  THEN 30
                   ELSE (
                    IF (the_month=5)
                      THEN 31
                       ELSE (
                        IF (the_month=6)
                          THEN 30
                           ELSE (
                            IF (the_month=7)
                              THEN 31
                               ELSE (
                                IF (the_month=8)
                                  THEN 31
                                   ELSE (
                                    IF (the_month=9)
                                      THEN 30
                                       ELSE (
                                        IF (the_month=10)
                                          THEN 31
                                           ELSE (
                                            IF (the_month=11)
                                              THEN 30
                                               ELSE
                                                31)))))))))));
--| number_of_months: (
    good_month(the_month)
  ->
    (
      (
```

```
        (
          (
            (
              (
                (
                  (
                    (
                      ((the_month=1) OR (the_month=2))
                     OR
                      (the_month=3))
                   OR
                    (the_month=4))
                 OR
                  (the_month=5))
               OR
                (the_month=6))
             OR
              (the_month=7))
           OR
            (the_month=8))
         OR
          (the_month=9))
       OR
        (the_month=10))
     OR
      (the_month=11))
   OR
    (the_month=12)));
--| days_in_month_range: (
    good_month(the_month)
  ->
    (
      (1<=days_in_month(the_month, the_year))
     AND
      (days_in_month(the_month, the_year)<=31)));
--| days_since_January_positive: (
    good_month(the_month)
  ->
    (0<=days_since_Jan(the_month, the_year)));
--| days_since_January_upper_bound: (
    (good_month(the_month) AND good_year(the_year))
  ->
    (days_since_Jan(the_month, the_year)<=335));
```

```
--| days_in_year_identity: (
    days_in_year(the_year)
  =
    (
      days_since_Jan(months_tic_last, the_year)
    +
      days_in_month(months_tic_last, the_year)));
--| END LEMMAS;
--| TRAIT time_representation_conversions IS
--| DECLARES
--| INTRODUCES
--| time_to_cal_time: time_sort -> calendar_time_sort;
--| cal_time_to_time: calendar_time_sort -> time_sort;
--| round_cal_time_down_to_milliseconds:
      calendar_time_sort -> calendar_time_sort;
--| interval_to_duration: interval_sort -> duration_sort;
--| round_duration_down_to_milliseconds:
      duration_sort -> time_sort;
--| duration_to_interval: duration_sort -> interval_sort;
--| AXIOMS: FORALL [the_time:time_sort,
the_calendar_time:calendar_time_sort,
the_interval:interval_sort, the_duration:duration_sort,
the_second:seconds]
--| time_to_cal_time: (
    time_to_cal_time(the_time)
  =
    (
      (seconds_since_1900(the_time.the_year)*seconds_to_catu)
    +
      (
        calendar_time_sort_identity(the_time.the_millisecond)
      *
        milliseconds_to_catu)));
--| round_cal_time_down: (
    round_cal_time_down_to_milliseconds(the_calendar_time)
  =
    (
      the_calendar_time
    -
      (the_calendar_time MOD milliseconds_to_catu)));
--| time_to_cal_time: (
    (
      time_to_cal_time(the_time)
    =
```

```
      round_cal_time_down_to_milliseconds(the_calendar_time))
  ->
    (cal_time_to_time(the_calendar_time)=the_time));
--| interval_to_duration: (
    interval_to_duration(the_interval)
  =
    (
      (
        (
          (
            seconds_to_duration(
              (the_interval.elapsed_days*days_to_seconds))
          +
            seconds_to_duration(
              (the_interval.elapsed_hours*hours_to_seconds)))
        +
          seconds_to_duration(
            (the_interval.elapsed_minutes*minutes_to_seconds)))
      +
        seconds_to_duration(the_interval.elapsed_seconds))
    +
      (
        seconds_to_duration(the_interval.elapsed_milliseconds)
      /
        seconds_to_milliseconds)));
--| good_interval_to_good_duration: (
    (
      good_interval(the_interval)
     AND
      (the_duration=interval_to_duration(the_interval)))
  ->
    good_duration(the_duration));
--| round_duration_down: (
    round_duration_down_to_milliseconds(the_duration)
  =
    (
      seconds_to_duration(
        Floor((the_duration*seconds_to_milliseconds)))
    /
      seconds_to_milliseconds));
--| good_seconds_to_good_duration: (
    good_second(the_second)
  ->
    good_duration(seconds_to_duration(the_second)));
```

```
--| END AXIOMS;
--| LEMMAS: FORALL  [d1:duration_sort, d2:duration_sort,
m:duration_sort, n:duration_sort, p:duration_sort]
--| good_duration_pair: (FORALL n:Int,
  d:Int::((d/=0)->good_duration(pair(n, d))));
--| good_duration_sum: (
    (good_duration(d1) AND good_duration(d2))
  ->
    good_duration((d1+d2)));
--| subtract_duration_equals: (FORALL m:duration_sort,
  n:duration_sort, p:duration_sort::((m=(n-p))->((m+p)=n)));
--| END LEMMAS;
PACKAGE builtin IS
  TYPE duration IS RECORD
    sign: integer;
    whole_part: integer;
    fraction: integer;
  END RECORD;
  TYPE string IS ARRAY(integer) OF character;
  FUNCTION duration_ize(seconds : IN integer) RETURN duration;
  --| WHERE
  --|      RETURN seconds_to_duration(seconds);
  --|      RETURN result SUCH THAT good_duration(result);
  --| END WHERE;

END builtin;

WITH builtin;
PACKAGE calendar IS
  TYPE time IS NEW integer;
  TYPE year_number IS NEW integer;
  TYPE month_number IS NEW integer;
  TYPE day_number IS NEW integer;
  TYPE day_duration IS NEW builtin.duration;
  FUNCTION clock RETURN time;
  --| WHERE
  --| END WHERE;

  FUNCTION year(date : IN time) RETURN year_number;
  --| WHERE
  --|      RETURN extract_year(date);
  --| END WHERE;

  FUNCTION month(date : IN time) RETURN month_number;
```

```
--| WHERE
--|      RETURN extract_month(date);
--| END WHERE;

FUNCTION day(date : IN time) RETURN day_number;
--| WHERE
--|      RETURN extract_day(date);
--| END WHERE;

FUNCTION seconds(date : IN time) RETURN day_duration;
--| WHERE
--|      RETURN extract_duration(date);
--|      RETURN result SUCH THAT good_duration(result);
--| END WHERE;

PROCEDURE split(date : IN time;
  year : OUT year_number;
  month : OUT month_number;
  day : OUT day_number;
  seconds : OUT day_duration);
--| WHERE
--|      OUT (year=extract_year(date));
--|      OUT (month=extract_month(date));
--|      OUT (day=extract_day(date));
--|      OUT (seconds=extract_duration(date));
--| END WHERE;

FUNCTION time_of(year : IN year_number;
  month : IN month_number;
  day : IN day_number;
  seconds : IN day_duration) RETURN time;
--| WHERE
--|      IN good_date(year, month, day, seconds);
--|      RETURN time_of(year, month, day, seconds);
--| END WHERE;

FUNCTION "+"(left : IN time;
  right : IN builtin.duration) RETURN time;
--| WHERE
--|      RETURN (left+right);
--| END WHERE;

FUNCTION "-"(left : IN time;
  right : IN builtin.duration) RETURN time;
```

```
    --| WHERE
    --|      RETURN (left-right);
    --| END WHERE;

    FUNCTION "<"(left, right : IN time) RETURN boolean;
    --| WHERE
    --|      RETURN (left<right);
    --| END WHERE;

    FUNCTION "<="(left, right : IN time) RETURN boolean;
    --| WHERE
    --|      RETURN (left<=right);
    --| END WHERE;

    FUNCTION ">"(left, right : IN time) RETURN boolean;
    --| WHERE
    --|      RETURN (left>right);
    --| END WHERE;

    FUNCTION ">="(left, right : IN time) RETURN boolean;
    --| WHERE
    --|      RETURN (left>=right);
    --| END WHERE;

    FUNCTION day_duration_ize(seconds : IN integer)
      RETURN day_duration;
    --| WHERE
    --|      RETURN seconds_to_duration(seconds);
    --|      RETURN result SUCH THAT good_duration(result);
    --| END WHERE;

END calendar;

WITH builtin;
PACKAGE integer_utilities IS
  TYPE number IS NEW integer;
  TYPE base IS NEW integer;
  TYPE numbers IS ARRAY(integer) OF number;
  FUNCTION min(left : IN number;
    right : IN number) RETURN number;
  --| WHERE
  --|      RETURN (IF (left<right) THEN left  ELSE right);
  --| END WHERE;
```

```
FUNCTION min(the_numbers : IN numbers) RETURN number;
--| WHERE
--| END WHERE;

FUNCTION max(left : IN number;
  right : IN number) RETURN number;
--| WHERE
--|      RETURN (IF (left>right) THEN left  ELSE right);
--| END WHERE;

FUNCTION max(the_numbers : IN numbers) RETURN number;
--| WHERE
--| END WHERE;

FUNCTION is_positive(the_number : IN number) RETURN boolean;
--| WHERE
--|      RETURN (the_number>0);
--| END WHERE;

FUNCTION is_natural(the_number : IN number) RETURN boolean;
--| WHERE
--|      RETURN (the_number>=0);
--| END WHERE;

FUNCTION is_negative(the_number : IN number) RETURN boolean;
--| WHERE
--|      RETURN (the_number<0);
--| END WHERE;

FUNCTION is_zero(the_number : IN number) RETURN boolean;
--| WHERE
--|      RETURN (the_number=0);
--| END WHERE;

FUNCTION is_odd(the_number : IN number) RETURN boolean;
--| WHERE
--|      RETURN ((the_number MOD 2)=1);
--| END WHERE;

FUNCTION is_even(the_number : IN number) RETURN boolean;
--| WHERE
--|      RETURN ((the_number MOD 2)=0);
--| END WHERE;
```

```
  FUNCTION image_of(the_number : IN number;
    with_the_base : IN base) RETURN builtin.string;
  --| WHERE
  --| END WHERE;

  FUNCTION value_of(the_image : IN builtin.string;
    with_the_base : IN base) RETURN number;
  --| WHERE
  --| END WHERE;

END integer_utilities;

WITH builtin;
PACKAGE duration_utilities IS
  TYPE number IS NEW builtin.duration;
  FUNCTION integer_part(the_number : IN number) RETURN integer;
  --| WHERE
  --|      IN good_duration(the_number);
  --|      RETURN IntegerPart(the_number);
  --| END WHERE;

  FUNCTION real_part(the_number : IN number) RETURN number;
  --| WHERE
  --|      IN good_duration(the_number);
  --|      RETURN RealPart(the_number);
  --|      RETURN result SUCH THAT good_duration(result);
  --| END WHERE;

  FUNCTION floor(the_number : IN number) RETURN integer;
  --| WHERE
  --|      IN good_duration(the_number);
  --|      RETURN Floor(the_number);
  --| END WHERE;

  FUNCTION ceiling(the_number : IN number) RETURN integer;
  --| WHERE
  --|      IN good_duration(the_number);
  --|      RETURN Ceiling(the_number);
  --| END WHERE;

END duration_utilities;

WITH builtin, calendar;
PACKAGE calendar_utilities IS
```

```
TYPE natural IS NEW integer;
TYPE string IS ARRAY(integer) OF character;
TYPE year IS NEW calendar.year_number;
TYPE month IS NEW integer;
TYPE day IS NEW integer;
TYPE hour IS NEW integer;
TYPE minute IS NEW integer;
TYPE second IS NEW integer;
TYPE millisecond IS NEW integer;
TYPE time IS RECORD
  the_year: year;
  the_month: month;
  the_day: day;
  the_hour: hour;
  the_minute: minute;
  the_second: second;
  the_millisecond: millisecond;
END RECORD;
TYPE interval IS RECORD
  elapsed_days: natural;
  elapsed_hours: hour;
  elapsed_minutes: minute;
  elapsed_seconds: second;
  elapsed_milliseconds: millisecond;
END RECORD;
TYPE year_day IS NEW integer;
TYPE month_name IS (january, february, march, april, may,
june, july, august, september, october, november, december);
TYPE day_name IS (monday, tuesday, wednesday, thursday,
friday, saturday, sunday);
TYPE period IS (am, pm);
TYPE time_format IS (full, military);
TYPE date_format IS (full, month_day_year);
FUNCTION natural_ize(m : IN integer) RETURN natural;
--| WHERE
--|       IN (0<=m);
--|       RETURN m;
--| END WHERE;

FUNCTION integer_ize(m : IN integer) RETURN integer;
--| WHERE
--|       RETURN m;
--| END WHERE;
```

```
FUNCTION year_day_ize(m : IN integer) RETURN year_day;
--| WHERE
--|       IN ((1<=m) AND (m<=366));
--|       RETURN m;
--| END WHERE;

FUNCTION day_ize(m : IN integer) RETURN day;
--| WHERE
--|       IN ((1<=m) AND (m<=31));
--|       RETURN m;
--| END WHERE;

FUNCTION hour_ize(m : IN integer) RETURN hour;
--| WHERE
--|       IN ((0<=m) AND (m<24));
--|       RETURN m;
--| END WHERE;

FUNCTION minute_ize(m : IN integer) RETURN minute;
--| WHERE
--|       IN ((0<=m) AND (m<60));
--|       RETURN m;
--| END WHERE;

FUNCTION second_ize(m : IN integer) RETURN second;
--| WHERE
--|       IN ((0<=m) AND (m<60));
--|       RETURN m;
--| END WHERE;

FUNCTION millisecond_ize(m : IN integer) RETURN millisecond;
--| WHERE
--|       IN ((0<=m) AND (m<1000));
--|       RETURN m;
--| END WHERE;

FUNCTION millisecond_ize(m : IN builtin.duration)
  RETURN millisecond;
--| WHERE
--|       RETURN Floor((1000*m));
--| END WHERE;

FUNCTION year_number_ize(the_year : IN year)
  RETURN calendar.year_number;
```

```
--| WHERE
--|       IN good_year(the_year);
--|       RETURN the_year;
--| END WHERE;

FUNCTION month_number_ize(the_month : IN month)
  RETURN calendar.month_number;
--| WHERE
--|       IN good_month(the_month);
--|       RETURN the_month;
--| END WHERE;

FUNCTION day_number_ize(the_day : IN day)
  RETURN calendar.day_number;
--| WHERE
--|       IN good_month_day(the_day);
--|       RETURN the_day;
--| END WHERE;

FUNCTION year_tic_first RETURN integer;
--| WHERE
--|       RETURN years_tic_first;
--| END WHERE;

FUNCTION month_name_tic_pos(the_month : IN month_name)
  RETURN month;
--| WHERE
--|       RETURN month_name_sort_tic_pos(the_month);
--| END WHERE;

FUNCTION month_name_tic_val(index : IN integer)
  RETURN month_name;
--| WHERE
--|       IN good_month((index+1));
--|       RETURN month_name_sort_tic_val(index);
--| END WHERE;

FUNCTION day_name_tic_pos(the_day : IN day_name)
  RETURN integer;
--| WHERE
--|       RETURN day_name_sort_tic_pos(the_day);
--| END WHERE;

FUNCTION day_name_tic_val(index : IN integer) RETURN day_name;
```

```
--| WHERE
--|      RETURN day_name_sort_tic_val(index);
--|      RAISE constraint_error <=> IN (
  (index<0)
 OR
  (index>6));
--| END WHERE;

FUNCTION millisecond_tic_first RETURN millisecond;
--| WHERE
--|      RETURN milliseconds_tic_first;
--| END WHERE;

FUNCTION second_tic_first RETURN second;
--| WHERE
--|      RETURN seconds_tic_first;
--| END WHERE;

FUNCTION minute_tic_first RETURN minute;
--| WHERE
--|      RETURN minutes_tic_first;
--| END WHERE;

FUNCTION hour_tic_first RETURN hour;
--| WHERE
--|      RETURN hours_tic_first;
--| END WHERE;

FUNCTION month_tic_first RETURN month;
--| WHERE
--|      RETURN months_tic_first;
--| END WHERE;

FUNCTION month_tic_last RETURN month;
--| WHERE
--|      RETURN months_tic_last;
--| END WHERE;

FUNCTION is_leap_year(the_year : IN year) RETURN boolean;
--| WHERE
--|      RETURN is_leap_year(the_year);
--| END WHERE;

FUNCTION days_in(the_year : IN year) RETURN year_day;
```

```
--| WHERE
--|       IN good_year(the_year);
--|       RETURN days_in_year(the_year);
--| END WHERE;

FUNCTION days_in(the_month : IN month;
  the_year : IN year) RETURN day;
--| WHERE
--|       IN good_month(the_month);
--|       IN good_year(the_year);
--|       RETURN days_in_month(the_month, the_year);
--| END WHERE;

FUNCTION month_of(the_month : IN month) RETURN month_name;
--| WHERE
--|       IN good_month(the_month);
--|       RETURN month_name_sort_tic_val((the_month-1));
--| END WHERE;

FUNCTION month_of(the_month : IN month_name) RETURN month;
--| WHERE
--|       RETURN (month_name_sort_tic_pos(the_month)+1);
--| END WHERE;

FUNCTION day_of(the_year : IN year;
  the_day : IN year_day) RETURN day_name;
--| WHERE
--|       IN good_year_and_day(the_year, the_day);
--|       RETURN day_name_of(the_year, the_day);
--| END WHERE;

FUNCTION day_of(the_time : IN time) RETURN year_day;
--| WHERE
--|       IN good_time(the_time);
--|       RETURN day_of_year(the_time.the_year,
  the_time.the_month, the_time.the_day);
--| END WHERE;

FUNCTION time_of(the_year : IN year;
  the_day : IN year_day) RETURN time;
--| WHERE
--|       IN good_year_and_day(the_year, the_day);
--|       RETURN t SUCH THAT (
  (
```

```
    (
      (
        (
          (
            (t.the_year=the_year)
           AND
            (t.the_month=month_of(the_year, the_day)))
         AND
          (t.the_day=day_of_month(the_year, the_day)))
       AND
        (t.the_hour=0))
     AND
      (t.the_minute=0))
   AND
    (t.the_second=0))
 AND
  (t.the_millisecond=0));
--| END WHERE;

FUNCTION period_of(the_time : IN time) RETURN period;
--| WHERE
--|      IN good_time(the_time);
--|      RETURN (IF (the_time.the_hour<12) THEN am  ELSE pm);
--| END WHERE;

FUNCTION time_of(the_time : IN time) RETURN calendar.time;
--| WHERE
--|      IN good_time(the_time);
--|      RETURN time_to_cal_time(the_time);
--| END WHERE;

FUNCTION time_of(the_time : IN calendar.time) RETURN time;
--| WHERE
--|      RETURN cal_time_to_time(the_time);
--| END WHERE;

FUNCTION duration_of(the_interval : IN interval)
  RETURN builtin.duration;
--| WHERE
--|      RETURN interval_to_duration(the_interval);
--|      RETURN result SUCH THAT good_duration(result);
--| END WHERE;

FUNCTION interval_of(the_duration : IN builtin.duration) RETURN
```

```
   interval;
  --| WHERE
  --|      RETURN duration_to_interval(the_duration);
  --| END WHERE;

END calendar_utilities;

WITH integer_utilities, duration_utilities, builtin;
PACKAGE BODY calendar_utilities IS
  TYPE month_day IS ARRAY(month) OF day;
  century_offset : Integer := 1900;
  days_per_year : year_day := 365;
  first_day : day_name := tuesday;
  seconds_per_minute : Integer := 60;
  seconds_per_hour : Integer := (60*seconds_per_minute);
  seconds_per_day : Integer := (24*seconds_per_hour);
  milliseconds_per_second : Integer := 1000;
  noon : Integer := 12;
  time_separator : Character := ':';
  date_separator : Character := '/';
  blank : Character := ' ';
  comma : Character := ',';
  zero : Character := '0';
  FUNCTION "+"(m, n : IN builtin.duration)
    RETURN builtin.duration;
  --| WHERE
  --|      IN good_duration(m);
  --|      IN good_duration(n);
  --|      RETURN (m+n);
  --|      RETURN result SUCH THAT good_duration(result);
  --| END WHERE;

  FUNCTION "-"(m, n : IN builtin.duration)
    RETURN builtin.duration;
  --| WHERE
  --|      IN good_duration(m);
  --|      IN good_duration(n);
  --|      RETURN (m-n);
  --|      RETURN result SUCH THAT good_duration(result);
  --| END WHERE;

  FUNCTION "*"(m, n : IN builtin.duration)
    RETURN builtin.duration;
  --| WHERE
```

```
--|      IN good_duration(m);
--|      IN good_duration(n);
--|      RETURN (m*n);
--|      RETURN result SUCH THAT good_duration(result);
--| END WHERE;

FUNCTION "/"(m, n : IN builtin.duration)
  RETURN builtin.duration;
--| WHERE
--|      IN good_duration(m);
--|      IN good_duration(n);
--|      IN (n/=Int_to_fixnum(0));
--|      RETURN (m/n);
--|      RETURN result SUCH THAT good_duration(result);
--| END WHERE;

FUNCTION "*"(m : IN builtin.duration;
  n : IN integer) RETURN builtin.duration;
--| WHERE
--|      IN good_duration(m);
--|      RETURN (m*n);
--|      RETURN result SUCH THAT good_duration(result);
--| END WHERE;

FUNCTION "/"(m : IN builtin.duration;
  n : IN integer) RETURN builtin.duration;
--| WHERE
--|      IN good_duration(m);
--|      IN (n/=0);
--|      RETURN (m*n);
--|      RETURN result SUCH THAT good_duration(result);
--| END WHERE;

FUNCTION days_per_month(the_month : IN month) RETURN integer;
--| WHERE
--|      IN ((1<=the_month) AND (the_month<=12));
--|      RETURN (
  (
    (
      (
        (
          (
            ((((((month_days_tic[1=>31])[2=>28])[3=>31])
```

```
                [4=>30])[5=>31])[6=>30])[7=>31])[8=>31])
                [9=>30])[10=>31])[11=>30])[12=>31])
                [the_month]);
  --| END WHERE;

  FUNCTION is_leap_year(the_year : IN year) RETURN Boolean
    --| WHERE * * *
    --|       RETURN is_leap_year(the_year);
    --| END WHERE;
    --! VC Status: proved
    --! BY instantiation of axiom leap_year
    --!       in trait time_representations
    --!       WITH the_year FOR the_year:Int
    --! BY instantiation of axiom non_leap_centennial
    --!       in trait time_representations
    --!       WITH the_year FOR the_year:Int
    --! BY instantiation of lemma mod_transitive in trait Z
    --!       WITH 100, 4, the_year FOR m1:Int, m2:Int, a:Int
    --! BY left substitution of 1
    --! BY left substitution of 2
    --! BY simplification
    --! BY synthesis of TRUE
    --! □

  IS


  BEGIN
    IF ((the_year MOD 100)=0) THEN
      RETURN  ((the_year MOD 400)=0);
    ELSE
      RETURN  ((the_year MOD 4)=0);
    END IF;
  END is_leap_year;
  FUNCTION days_in(the_year : IN year) RETURN year_day
    --| WHERE
    --|       GLOBAL days_per_year : IN ;
    --|       IN (days_per_year=365);
    --|       RETURN days_in_year(the_year);
    --| END WHERE;
    --! VC Status: proved
    --! BY axiom days_in_year in trait time_representations
    --!       WITH the_year FOR the_year:Int
    --!       substituting for left
```

```
   --! BY simplification
   --! BY synthesis of TRUE
   --! □

 IS


 BEGIN
   IF is_leap_year(the_year) THEN
     RETURN  (days_per_year+1);
   ELSE
     RETURN  days_per_year;
   END IF;
 END days_in;
 FUNCTION days_in(the_month : IN month;
   the_year : IN year) RETURN day
   --| WHERE * * *
   --|       IN good_month(the_month);
   --|       IN good_year(the_year);
   --|       RETURN days_in_month(the_month, the_year);
   --| END WHERE;
   --! VC Status: proved
   --! BY axiom month_pos_february in trait sort_names
   --!       WITH  FOR
   --!       substituting for left
   --! BY instantiation of lemma number_of_months
           in trait time_representations
   --!       WITH the_month FOR the_month:Int
   --! BY analysis of IMPLIES, in 3
   --!    BY hypothesis
   --! AND THEN
   --!    BY contradiction, in 1
   --!    BY axiom good_month in trait time_representations
   --!         WITH the_month FOR the_month:Int
   --!         substituting for left
   --!    BY contradiction, in 1
   --!    BY lemma days_in_months in trait time_representations
   --!         WITH the_month, the_year FOR the_month:Int,
     the_year:Int
   --!         substituting for left
   --!    BY cases, using (the_month=1)
   --!    CASE TRUE
   --!      BY left substitution of 4
   --!      BY synthesis of TRUE
```

```
--!    CASE FALSE
--!      BY cases, using (the_month=2)
--!      CASE TRUE
--!        BY left substitution of 5
--!        BY axiom days_in_February
                in trait time_representations
--!             WITH the_year FOR the_year:Int
--!             substituting for left
--!        BY simplification
--!        BY synthesis of TRUE
--!      CASE FALSE
--!        BY simplification
--!        BY cases, using (the_month=3)
--!        CASE TRUE
--!          BY left substitution of 4
--!          BY synthesis of TRUE
--!        CASE FALSE
--!          BY cases, using (the_month=4)
--!          CASE TRUE
--!            BY left substitution of 5
--!            BY synthesis of TRUE
--!          CASE FALSE
--!            BY cases, using (the_month=5)
--!            CASE TRUE
--!              BY left substitution of 6
--!              BY synthesis of TRUE
--!            CASE FALSE
--!              BY cases, using (the_month=6)
--!              CASE TRUE
--!                BY left substitution of 7
--!                BY synthesis of TRUE
--!              CASE FALSE
--!                BY cases, using (the_month=7)
--!                CASE TRUE
--!                  BY left substitution of 8
--!                  BY synthesis of TRUE
--!                CASE FALSE
--!                  BY cases, using (the_month=8)
--!                  CASE TRUE
--!                    BY left substitution of 9
--!                    BY synthesis of TRUE
--!                  CASE FALSE
--!                    BY cases, using (the_month=9)
--!                    CASE TRUE
```

```
  --!                    BY left substitution of 10
  --!                    BY synthesis of TRUE
  --!                  CASE FALSE
  --!                    BY cases, using (the_month = 10)
  --!                    CASE TRUE
  --!                      BY left substitution of 11
  --!                      BY synthesis of TRUE
  --!                    CASE FALSE
  --!                      BY cases, using (the_month = 11)
  --!                      CASE TRUE
  --!                        BY left substitution of 12
  --!                        BY synthesis of TRUE
  --!                      CASE FALSE
  --!                        BY cases, using (the_month = 12)
  --!                        CASE TRUE
  --!                          BY left substitution of 13
  --!                          BY synthesis of TRUE
  --!                        CASE FALSE
  --!                          BY array simplification
  --!                          BY simplification
  --!                          BY synthesis of TRUE
  --! □

IS


BEGIN
  IF (the_month=(month_name_tic_pos(february)+1)) THEN
    IF is_leap_year(the_year) THEN
      RETURN  (days_per_month((month_name_tic_pos(february)+1))
    +1);
    ELSE
      RETURN  days_per_month(the_month);
    END IF;
  ELSE
    RETURN  days_per_month(the_month);
  END IF;
END days_in;
FUNCTION month_of(the_month : IN month) RETURN month_name
  --| WHERE * * *
  --|       IN good_month(the_month);
  --|       RETURN month_name_sort_tic_val((the_month-1));
  --| END WHERE;
  --! VC Status: proved
```

```
  --! BY synthesis of TRUE
  --! □

IS


BEGIN
  RETURN  month_name_tic_val((the_month-1));
END month_of;
FUNCTION month_of(the_month : IN month_name) RETURN month
  --| WHERE * * *
  --|       RETURN (month_name_sort_tic_pos(the_month)+1);
  --| END WHERE;
  --! VC Status: proved
  --! BY synthesis of TRUE
  --! □

IS


BEGIN
  RETURN  (month_name_tic_pos(the_month)+1);
END month_of;
FUNCTION day_of(the_year : IN year;
  the_day : IN year_day) RETURN day_name
  --| WHERE
  --|       GLOBAL first_day : IN ;
  --|       IN (first_day=tuesday);
  --|       IN good_year_and_day(the_year, the_day);
  --|       RETURN day_name_of(the_year, the_day);
  --| END WHERE;
  --! VC Status: proved
  --! BY contradiction, in 2
  --! BY axiom good_year_and_day in trait time_representations
  --!       WITH the_year, the_day FOR the_year:Int,
    the_year_day:Int
  --!       substituting for left
  --! BY contradiction, in 2
  --! BY axiom first_year in trait sort_names
  --!       WITH  FOR
  --!       substituting for left
  --! BY simplification
  --! BY axiom day_name in trait time_representations
  --!       WITH 1901, 1 FOR the_year:Int, the_year_day:Int
```

```
  --!      substituting for left
  --! BY axiom days_since_1900_1 in trait time_representations
  --!      WITH 1901 FOR the_year:Int
  --!      substituting for left
  --! BY axiom days_since_1900_0 in trait time_representations
  --!      WITH  FOR
  --!      substituting for left
  --! BY claiming (days_in_year(1900)=365)
  --!    BY axiom days_in_year in trait time_representations
  --!         WITH 1900 FOR the_year:Int
  --!         substituting for left
  --!    BY instantiation of axiom leap_year
  --!         in trait time_representations
  --!         WITH 1900 FOR the_year:Int
  --!    BY instantiation of axiom non_leap_centennial in trait
   time_representations
  --!         WITH 1900 FOR the_year:Int
  --!    BY left substitution of 6
  --!    BY left substitution of 7
  --!    BY synthesis of TRUE
  --! THEN
  --!    BY instantiation of axiom day_val_1
  --!         in trait sort_names
  --!         WITH  FOR
  --!    BY simplification
  --!    BY synthesis of TRUE
  --! □

IS
  result : day_name := first_day;
  index : integer;
  PROCEDURE increment(the_day : IN OUT day_name;
    offset : IN natural)
    --| WHERE
    --|      IN good_day_name(the_day);
    --|      IN (offset>=0);
    --|      IN (offset<=6);
    --|      OUT (
                the_day
               =
                day_name_sort_increment(IN the_day, offset));
    --|      OUT good_day_name(the_day);
    --| END WHERE;
    --! VC Status: proved
```

```
--! BY simplification
--! BY instantiation of lemma day_name_sort_tic_pos_range
        in trait sort_names
--!      WITH the_day FOR the_day_name:day_name_sort
--! BY simplification
--! BY axiom day_name_sort_increment in trait sort_names
--!      WITH the_day, offset FOR
  the_day_name:day_name_sort, offset:Int
--!      substituting for left
--! BY instantiation of lemma mod_lower in trait Z
--!      WITH 7, (day_name_sort_tic_pos(the_day)+offset)
  FOR m:Int, a:Int
--! BY instantiation of lemma mod_upper in trait Z
--!      WITH 7, (day_name_sort_tic_pos(the_day)+offset)
  FOR m:Int, a:Int
--! BY axiom good_day_name in trait sort_names
--!      WITH
  day_name_sort_tic_val(
    ((day_name_sort_tic_pos(the_day)+offset) MOD 7)) FOR
  the_day_name:day_name_sort
--!      substituting for left
--! BY instantiation of lemma day_name_sort_tic_val_range
        in trait sort_names
--!      WITH
  ((day_name_sort_tic_pos(the_day)+offset) MOD 7) FOR
  the_pos:Int
--! BY simplification
--! BY instantiation of lemma mod_subtract in trait Z
--!      WITH 7, (day_name_sort_tic_pos(the_day)+offset)
  FOR m:Int, a:Int
--! BY simplification
--! BY cases, using (
  (day_name_sort_tic_pos(the_day)+offset)
<
  7)
--! CASE TRUE
--!    BY lemma mod_ident in trait Z
--!         WITH (day_name_sort_tic_pos(the_day)+offset),
  7 FOR a:Int, m:Int
--!         substituting for left
--!    BY simplification
--!    BY synthesis of TRUE
--! CASE FALSE
--!    BY simplification
```

```
    --!    BY synthesis of TRUE
    --! □

  IS


  BEGIN
    the_day:=day_name_tic_val((day_name_tic_pos(the_day)+
      offset));
  EXCEPTION
    WHEN constraint_error =>
      the_day:=day_name_tic_val((
        (day_name_tic_pos(the_day)+offset)-7));
  END increment;
BEGIN
  index:=(year_tic_first()+1);
  --! VC Status: ** not proved **
  --! BY simplification
  --! BY instantiation of axiom day_name
  --!         in trait time_representations
  --!      WITH (index-1), 1 FOR the_year:Int, the_year_day:Int
  --! BY instantiation of lemma day_name_sort_tic_val_range in
  trait sort_names
  --!      WITH
    ((days_of_years_since_1900((index-1))+1) MOD 7) FOR
    the_pos:Int
  --! BY instantiation of lemma mod_upper in trait Z
  --!      WITH 7, (days_of_years_since_1900((index-1))+1) FOR
    m:Int, a:Int
  --! BY instantiation of lemma mod_lower in trait Z
  --!      WITH 7, (days_of_years_since_1900((index-1))+1) FOR
    m:Int, a:Int
  --! BY axiom good_day_name in trait sort_names
  --!      WITH result FOR the_day_name:day_name_sort
  --!      substituting for left
  --! BY simplification
  --! BY synthesis of IF,
  --!    1. (years_tic_first<index)
  --!    2. (index<=(1+the_year))
  --!    3. (result=day_name_of((index-1), 1))
  --!    4. (result
    =

    day_name_sort_tic_val((days_of_years_since_1900((index-1))
```

```
      MOD
       7)))
--!    5. (0
  <=
  ((1+days_of_years_since_1900((index-1))) MOD 7))
--!    6. (((1+days_of_years_since_1900((index-1))) MOD 7)
  <=
  6)
--!    7. (monday
  <=

  day_name_sort_tic_val(((1
        +
        days_of_years_since_1900((index-1)))
      MOD
       7)))
--!    8. (
  day_name_sort_tic_val(((1
        +
        days_of_years_since_1900((index-1)))
      MOD
       7))
  <=
  sunday)
--!    9. (index<=the_year)
--!    >>
  (IF is_leap_year(the_year)
    THEN (good_day_name(day_name_sort_increment(result, 2))
      ->
      (day_name_sort_increment(result, 2)
        =
        day_name_of(index, 1)))
    ELSE (good_day_name(day_name_sort_increment(result, 1))
      ->
      (day_name_sort_increment(result, 1)
        =
        day_name_of(index, 1))))
--!    <proof>
--! AND
--!    BY instantiation of lemma mod_lower in trait Z
--!         WITH 7, the_day FOR m:Int, a:Int
--!    BY instantiation of lemma mod_upper in trait Z
--!         WITH 7, the_day FOR m:Int, a:Int
--!    BY simplification
```

```
--!    1. (years_tic_first<index)
--!    2. (index<=(1+the_year))
--!    3. (result=day_name_of((index-1), 1))
--!    4. (result
  =

  day_name_sort_tic_val((days_of_years_since_1900((index-1))
      MOD
       7)))
--!    5. (0
  <=
  ((1+days_of_years_since_1900((index-1))) MOD 7))
--!    6. (((1+days_of_years_since_1900((index-1))) MOD 7)
  <=
  6)
--!    7. (monday
  <=

  day_name_sort_tic_val(((1
       +
       days_of_years_since_1900((index-1)))
      MOD
       7)))
--!    8. (
  day_name_sort_tic_val(((1
       +
       days_of_years_since_1900((index-1)))
      MOD
       7))
  <=
  sunday)
--!    9. (index>the_year)
--!    10. (0<=(the_day MOD 7))
--!    11. ((the_day MOD 7)<=6)
--!    >> ((0<(the_day MOD 7))
  AND
  (
    good_day_name(
      day_name_sort_increment(result, ((the_day MOD 7)-1)))
    ->
    (day_name_sort_increment(result, ((the_day MOD 7)-1))
      =
      day_name_of(the_year, the_day))))
--!    <proof>
```

```
  --! □
  WHILE (index<=the_year) LOOP
    --| INVARIANT  (
        (
          ((years_tic_first+1)<=index)
         AND
          (index<=(the_year+1)))
       AND
        (result=day_name_of((index-1), 1)));
    IF is_leap_year(the_year) THEN
      increment(result, 2);
    ELSE
      increment(result, 1);
    END IF;
    index:=(index+1);
   END LOOP;
  increment(result, natural_ize(((the_day MOD 7)-1)));
  RETURN  result;
END day_of;
FUNCTION day_of(the_time : IN time) RETURN year_day
  --| WHERE * * *
  --|       IN good_time(the_time);
  --|       RETURN day_of_year(the_time.the_year,
    the_time.the_month, the_time.the_day);
  --| END WHERE;
  --! VC Status: proved
  --! BY simplification
  --! BY instantiation of axiom days_since_January_0
          in trait time_representations
  --!       WITH the_time.the_year FOR the_year:Int
  --! BY instantiation of axiom months_tic_first
          in trait sort_names
  --!       WITH  FOR
  --! BY simplification
  --! BY simplification
  --! BY contradiction, in 1
  --! BY axiom good_time in trait time_representations
  --!       WITH the_time FOR the_time:AnyRecordSort
  --!       substituting for left
  --! BY contradiction, in 1
  --! BY simplification
  --! BY synthesis of TRUE
  --! □
```

```
IS
  result : natural := 0;
  index : month;

BEGIN
  index:=month_tic_first();
  --! VC Status: proved
  --! BY simplification
  --! BY contradiction, in 2
  --! BY axiom good_time in trait time_representations
  --!       WITH the_time FOR the_time:AnyRecordSort
  --!       substituting for left
  --! BY contradiction, in 2
  --! BY simplification
  --! BY synthesis of IF,
  --!     BY axiom days_since_January_1
              in trait time_representations
  --!         WITH (1+index), the_time.the_year FOR the_month:
    Int, the_year:Int
  --!         substituting for left
  --!     BY instantiation of lemma days_in_month_range in trait
   time_representations
  --!         WITH index, the_time.the_year FOR the_month:Int,
    the_year:Int
  --!     BY simplification
  --!     BY axiom good_month in trait time_representations
  --!         WITH index FOR the_month:Int
  --!         substituting for left
  --!     BY instantiation of axiom months_tic_first
              in trait sort_names
  --!         WITH  FOR
  --!     BY simplification
  --!     BY simplification
  --!     BY axiom good_year in trait time_representations
  --!         WITH the_time.the_year FOR the_year:Int
  --!         substituting for left
  --!     BY simplification
  --!     BY synthesis of TRUE
  --! AND
  --!     BY instantiation of lemma days_since_January_positive
in trait time_representations
  --!         WITH index, the_time.the_year FOR the_month:Int,
    the_year:Int
  --!     BY simplification
```

```
  --!    BY axiom day_of_year in trait time_representations
  --!         WITH the_time.the_year, index, the_time.the_day
    FOR the_year:Int, the_month:Int, the_month_day:Int
  --!         substituting for left
  --!    BY simplification
  --!    BY instantiation
              of lemma days_since_January_upper_bound
              in trait time_representations
  --!         WITH index, the_time.the_year FOR the_month:Int,
    the_year:Int
  --!    BY instantiation of axiom good_month
              in trait time_representations
  --!         WITH index FOR the_month:Int
  --!    BY instantiation of axiom good_year
              in trait time_representations
  --!         WITH the_time.the_year FOR the_year:Int
  --!    BY instantiation of lemma days_in_month_range in trait
   time_representations
  --!         WITH the_time.the_month, the_time.the_year FOR
    the_month:Int, the_year:Int
  --!    BY simplification
  --!    BY synthesis of TRUE
  --! □
  WHILE (index<=(the_time.the_month-1)) LOOP
    --| INVARIANT  (
        (
          (
            (result=days_since_Jan(index, the_time.the_year))
           AND
            good_time(the_time))
         AND
          (months_tic_first<=index))
       AND
        (index<=the_time.the_month));
    result:=(result+
      natural_ize(days_in(index, the_time.the_year)));
    index:=(index+1);
   END LOOP;
  RETURN  year_day_ize((result+natural_ize(the_time.the_day)));
END day_of;
FUNCTION time_of(the_year : IN year;
  the_day : IN year_day) RETURN time
  --| WHERE * * *
  --|       IN good_year_and_day(the_year, the_day);
```

```
--|      RETURN t SUCH THAT (
  (
    (
      (
        (
          (
            (t.the_year=the_year)
           AND
            (t.the_month=month_of(the_year, the_day)))
         AND
          (t.the_day=day_of_month(the_year, the_day)))
       AND
        (t.the_hour=0))
     AND
      (t.the_minute=0))
   AND
    (t.the_second=0))
 AND
  (t.the_millisecond=0));
--| END WHERE;
--! VC Status: proved
--! BY axiom months_tic_first in trait sort_names
--!      WITH  FOR
--!      substituting for left
--! BY axiom months_tic_last in trait sort_names
--!      WITH  FOR
--!      substituting for left
--! BY contradiction, in 1
--! BY axiom good_year_and_day in trait time_representations
--!      WITH the_year, the_day FOR the_year:Int,
  the_year_day:Int
--!      substituting for left
--! BY contradiction, in 1
--! BY simplification
--! BY axiom good_year in trait time_representations
--!      WITH the_year FOR the_year:Int
--!      substituting for left
--! BY simplification
--! BY axiom good_month in trait time_representations
--!      WITH 1 FOR the_month:Int
--! BY lemma days_in_month_range
        in trait time_representations
--! BY analysis of FORALL, in 5
--!      WITH 1, the_year FOR the_month:Int, the_year:Int
```

```
  --! BY simplification
  --! BY axiom days_since_January_0
          in trait time_representations
  --!     WITH the_year FOR the_year:Int
  --!     substituting for left
  --! BY instantiation of lemma days_in_year_identity
          in trait time_representations
  --!     WITH the_year FOR the_year:Int
  --! BY instantiation of axiom months_tic_last
          in trait sort_names
  --!     WITH  FOR
  --! BY simplification
  --! BY synthesis of TRUE
  --! □

IS
  result : year_day := the_day;
  temp : time;
  index : month;

BEGIN
  index:=month_tic_first();
  --! VC Status: proved
  --! BY simplification
  --! BY instantiation of lemma days_in_month_range
         in trait time_representations
  --!    WITH index, the_year FOR the_month:Int, the_year:Int
  --! BY simplification
  --! BY axiom good_month in trait time_representations
  --!    WITH index FOR the_month:Int
  --!    substituting for left
  --! BY simplification
  --! BY axiom months_tic_first in trait sort_names
  --! BY axiom months_tic_last in trait sort_names
  --! BY simplification
  --! BY simplification
  --! BY synthesis of IF,
  --!   BY instantiation of axiom month_of
           in trait time_representations
  --!      WITH index, the_year, the_day FOR the_month:Int,
    the_year:Int, the_year_day:Int
  --!   BY simplification
  --!   BY axiom day_of_month in trait time_representations
  --!      WITH the_year, the_day FOR the_year:Int,
```

```
  the_year_day:Int
--!          substituting for left
--!    BY simplification
--!    BY axiom first_hour in trait sort_names
--!    BY axiom first_minute in trait sort_names
--!    BY axiom first_second in trait sort_names
--!    BY axiom first_millisecond in trait sort_names
--!    BY simplification
--!    BY simplification
--!    BY synthesis of TRUE
--! AND
--!    BY instantiation of axiom days_since_January_1
--!         in trait time_representations
--!         WITH (1+index), the_year FOR the_month:Int,
  the_year:Int
--!    BY simplification
--!    BY synthesis of TRUE
--! □
WHILE (index<=month_tic_last()) LOOP
  --| INVARIANT  (
      (
        (
          ((days_since_Jan(index, the_year)+result)=the_day)
         AND
          (
            (
              (months_tic_first<=index)
             AND
              (index<=months_tic_last))
            AND
             good_year(the_year)))
       AND
        (result>0))
     AND
      (
        the_day
       <=
        (
          days_since_Jan(months_tic_last, the_year)
        +
          days_in_month(months_tic_last, the_year))));
 IF (result<=year_day_ize(days_in(index, the_year))) THEN
   temp.the_year:=the_year;
   temp.the_month:=index;
```

```
      temp.the_day:=day_ize(result);
      temp.the_hour:=hour_tic_first();
      temp.the_minute:=minute_tic_first();
      temp.the_second:=second_tic_first();
      temp.the_millisecond:=millisecond_tic_first();
      RETURN  temp;
    ELSE
      result:=(result-year_day_ize(days_in(index, the_year)));
    END IF;
    index:=(index+1);
   END LOOP;
  RAISE lexical_error;
END time_of;
FUNCTION period_of(the_time : IN time) RETURN period
  --| WHERE
  --|      GLOBAL noon : IN ;
  --|      IN (noon=12);
  --|      IN good_time(the_time);
  --|      RETURN (IF (the_time.the_hour<12) THEN am  ELSE pm);
  --| END WHERE;
  --! VC Status: proved
  --! BY simplification
  --! BY synthesis of TRUE
  --! □

IS


BEGIN
  IF (the_time.the_hour>=noon) THEN
    RETURN  pm;
  ELSE
    RETURN  am;
  END IF;
END period_of;
FUNCTION time_of(the_time : IN time) RETURN calendar.time
  --| WHERE
  --|      GLOBAL seconds_per_hour, seconds_per_minute,
  milliseconds_per_second : IN ;
  --|      IN good_time(the_time);
  --|      RETURN time_to_cal_time(the_time);
  --| END WHERE;
  --! VC Status: ** not proved **
  --! BY contradiction, in 1
```

```
--! BY axiom good_time in trait time_representations
--!       WITH the_time FOR the_time:AnyRecordSort
--!       substituting for left
--! BY contradiction, in 1
--! BY analysis of AND, in 1
--! BY instantiation of lemma days_in_month_range
          in trait time_representations
--!       WITH the_time.the_month, the_time.the_year FOR
  the_month:Int, the_year:Int
--! BY simplification
--! BY axiom good_year in trait time_representations
--!       WITH the_time.the_year FOR the_year:Int
--!       substituting for left
--! BY axiom good_month in trait time_representations
--!       WITH the_time.the_month FOR the_month:Int
--!       substituting for left
--! BY axiom good_month_day in trait time_representations
--!       WITH the_time.the_day FOR the_month_day:Int
--!       substituting for left
--! BY simplification
--! BY claiming (
  days_in_month(the_time.the_month, the_time.the_year)
<=
  31)
--!     BY hypothesis
--! THEN
--!     BY simplification
--!     1. (1901<=(the_time.the_year))
--!     2. ((the_time.the_year)<=2099)
--!     3. (0<(the_time.the_month))
--!     4. ((the_time.the_month)<=12)
--!     5. (0<(the_time.the_day))
--!     6. ((the_time.the_day)
  <=
  days_in_month((the_time.the_month), (the_time.the_year)))
--!     7. (0<=(the_time.the_hour))
--!     8. ((the_time.the_hour)<days_to_hours)
--!     9. (0<=(the_time.the_minute))
--!     10. ((the_time.the_minute)<hours_to_minutes)
--!     11. (0<=(the_time.the_second))
--!     12. ((the_time.the_second)<minutes_to_seconds)
--!     13. (0<=(the_time.the_millisecond))
--!     14. ((the_time.the_millisecond)
  <
```

```
  seconds_to_milliseconds)
--!    15. (
  days_in_month((the_time.the_month), (the_time.the_year))
  <=
  31)
--!    16.
good_duration(
  seconds_to_duration(((the_time.the_millisecond)
      /
      milliseconds_per_second)))
--!    17.
good_duration(seconds_to_duration((the_time.the_second)))
--!    18.
good_duration(
  seconds_to_duration(((the_time.the_minute)
      *
      seconds_per_minute)))
--!    19.
good_duration(
  seconds_to_duration(((the_time.the_hour)
      *
      seconds_per_hour)))
--!    20.
good_duration((
    seconds_to_duration(((the_time.the_minute)
        *
        seconds_per_minute))
    +

    seconds_to_duration(((the_time.the_hour)
        *
        seconds_per_hour))))
--!    21.
good_duration(((
      seconds_to_duration(((the_time.the_hour)
          *
          seconds_per_hour))
      +

      seconds_to_duration(((the_time.the_minute)
          *
          seconds_per_minute)))
    +
    seconds_to_duration((the_time.the_second))))
```

```
--!    22.
good_duration((((
        seconds_to_duration(((the_time.the_hour)
            *
            seconds_per_hour))
        +
        seconds_to_duration((the_time.the_second)))
      +

      seconds_to_duration(((the_time.the_minute)
          *
          seconds_per_minute)))
    +

    seconds_to_duration(((the_time.the_millisecond)
        /
        milliseconds_per_second))))
--!    >> (
  good_date((the_time.the_year),
    (the_time.the_month),
    (the_time.the_day),
    (((
          seconds_to_duration(((the_time.the_hour)
              *
              seconds_per_hour))
          +
          seconds_to_duration((the_time.the_second)))
        +

        seconds_to_duration(((the_time.the_minute)
            *
            seconds_per_minute)))
      +

      seconds_to_duration(((the_time.the_millisecond)
          /
          milliseconds_per_second))))
  AND
  (
    time_of((the_time.the_year),
      (the_time.the_month),
      (the_time.the_day),
      (((
            seconds_to_duration(((the_time.the_hour)
```

```
                *
                seconds_per_hour))
            +
            seconds_to_duration((the_time.the_second)))
          +

          seconds_to_duration(((the_time.the_minute)
              *
              seconds_per_minute)))
        +

        seconds_to_duration(((the_time.the_millisecond)
            /
            milliseconds_per_second))))
    =
    time_to_cal_time(the_time)))
  --!    <proof>
  --! □

IS


BEGIN
  RETURN  calendar.time_of(year_number_ize(the_time.the_year),
  month_number_ize(the_time.the_month),
  day_number_ize(the_time.the_day),
  (
    (
      (
        calendar.day_duration_ize((
          integer_ize(the_time.the_hour)*
          seconds_per_hour))+
        calendar.day_duration_ize((
          integer_ize(the_time.the_minute)*
          seconds_per_minute)))+
      calendar.day_duration_ize(the_time.the_second))+
    calendar.day_duration_ize((the_time.the_millisecond/
      milliseconds_per_second))));
END time_of;
FUNCTION time_of(the_time : IN calendar.time) RETURN time
  --| WHERE
  --|      GLOBAL seconds_per_hour, seconds_per_minute,
  milliseconds_per_second : IN ;
  --|      RETURN cal_time_to_time(the_time);
```

```
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  result : time;
  total_duration : calendar.day_duration;
  seconds : natural;
  the_year : year;
  the_month : month;
  the_day : day;

BEGIN
  calendar.split(the_time,
  the_year,
  the_month,
  the_day,
  total_duration);
  -- To mollify our aggressive aliasing checks;
  result.the_year:=the_year;
  result.the_month:=the_month;
  result.the_day:=the_day;
  --! PRECONDITION = (good_duration(total_duration)
      AND
      (((0<=(Floor(total_duration)/seconds_per_hour))
          AND
          ((Floor(total_duration)/seconds_per_hour)<24))
        AND
        (((0
              <=
              ((Floor(total_duration) MOD seconds_per_hour)
                /
                seconds_per_minute))
            AND
            (((Floor(total_duration) MOD seconds_per_hour)
                /
                seconds_per_minute)
              <
              60))
          AND
          (((0
                <=
                ((Floor(total_duration) MOD seconds_per_hour)
                  MOD
```

```
                    seconds_per_minute))
                AND
                (((Floor(total_duration) MOD seconds_per_hour)
                    MOD
                     seconds_per_minute)
                  <
                  60))
              AND
              (good_duration(total_duration)
                AND
                ((good_duration(RealPart(total_duration))
                    AND

                    good_duration((RealPart(total_duration)
                        *
                        milliseconds_per_second)))
                  ->
                  (((((result[.the_hour
                            =>(Floor(total_duration)
                              /
                              seconds_per_hour)])[.the_minute
                          =>((Floor(total_duration)
                              MOD
                               seconds_per_hour)
                             /
                             seconds_per_minute)])[.the_second
                        =>((Floor(total_duration)
                            MOD
                             seconds_per_hour)
                           MOD
                            seconds_per_minute)])[.the_millisecond
                     =>
                     Floor((1000
                         *
                         (RealPart(total_duration)
                           *
                           milliseconds_per_second)))])
                   =
                   cal_time_to_time(the_time))))))));
seconds:=duration_utilities.floor(total_duration);
result.the_hour:=hour_ize((seconds/seconds_per_hour));
seconds:=(seconds MOD seconds_per_hour);
result.the_minute:=minute_ize((seconds/seconds_per_minute));
result.the_second:=second_ize((seconds MOD
```

```
    seconds_per_minute));
  result.the_millisecond:=millisecond_ize((
    duration_utilities.real_part(total_duration)*
    milliseconds_per_second));
  RETURN  result;
END time_of;
FUNCTION duration_of(the_interval : IN interval) RETURN builtin
.duration
  --| WHERE
  --|       GLOBAL seconds_per_day, seconds_per_hour,
  seconds_per_minute : IN ;
  --|       IN (seconds_per_hour=hours_to_seconds);
  --|       IN (seconds_per_minute=minutes_to_seconds);
  --|       IN (seconds_per_day=days_to_seconds);
  --|       IN good_interval(the_interval);
  --|       RETURN interval_to_duration(the_interval);
  --|       RETURN result SUCH THAT good_duration(result);
  --| END WHERE;
  --! VC Status: proved
  --! BY axiom Int_identity in trait Z
  --!       WITH (the_interval.elapsed_days*days_to_seconds)
    FOR n:Int
  --!       substituting for left
  --! BY lemma seconds_to_duration in trait sort_names
  --!       WITH (the_interval.elapsed_days*seconds_per_day)
    FOR the_second:Int
  --!       substituting for left
  --! BY simplification
  --! BY instantiation of lemma good_duration_pair
          in trait time_representation_conversions
  --!       WITH (seconds_per_day*the_interval.elapsed_days), 1
    FOR n:Int, d:Int
  --! BY simplification
  --! BY axiom good_second in trait sort_names
  --!       WITH (the_interval.elapsed_milliseconds/1000) FOR
    the_second:Int
  --!       substituting for left
  --! BY instantiation of axiom good_interval
          in trait sort_names
  --!       WITH the_interval FOR the_interval:AnyRecordSort
  --! BY contradiction, in 4
  --! BY left substitution of 7
  --! BY contradiction, in 4
  --! BY analysis of AND, in 4
```

```
  --! BY instantiation of axiom good_millisecond
          in trait sort_names
  --!       WITH the_interval.elapsed_milliseconds FOR
    the_millisecond:Int
  --! BY contradiction, in 11
  --! BY left substitution of 12
  --! BY contradiction, in 11
  --! BY claiming ((the_interval.elapsed_milliseconds/1000)=0)
  --!     BY lemma div_upper in trait Z
  --!          WITH the_interval.elapsed_milliseconds, 1000 FOR
    a:Int, m:Int
  --!          substituting for left
  --!     BY hypothesis
  --! THEN
  --!     BY simplification
  --!     BY synthesis of TRUE
  --! □

IS
  temp : builtin.duration;

BEGIN
  temp:=builtin.duration_ize((
    integer_ize(the_interval.elapsed_days)*seconds_per_day));
  --! VC Status: proved
  --! BY lemma pair_minus in trait fixed_point
  --!       WITH
    (
      Int_identity((the_interval.elapsed_days*days_to_seconds))
    +
      Int_identity(
        (the_interval.elapsed_hours*hours_to_seconds))), 1,
    (the_interval.elapsed_hours*seconds_per_hour), 1 FOR n1:
    Int, d1:Int, n2:Int, d2:Int
  --!       substituting for left
  --! BY axiom Int_identity in trait Z
  --!       WITH
    (
      Int_identity((the_interval.elapsed_days*days_to_seconds))
    +
      Int_identity(
        (the_interval.elapsed_hours*hours_to_seconds))) FOR n:
    Int
  --!       substituting for left
```

```
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_hours*seconds_per_hour)
  FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_hours*hours_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_days*days_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY contradiction, in 1
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_days*days_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY contradiction, in 1
--! BY simplification
--! BY synthesis of TRUE
--! □
--| ASSERT (
  (
    (
      temp
    =
      pair(
        Int_identity(
          (the_interval.elapsed_days*days_to_seconds)), 1))
   AND
    good_interval(the_interval))
 AND
  (
    (
      good_duration(temp)
     AND
      (
        good_interval(the_interval)
       AND
        good_second(
          (the_interval.elapsed_milliseconds/1000))))
   AND
    (
      (seconds_per_hour=hours_to_seconds)
```

```
     AND
      (seconds_per_minute=minutes_to_seconds))));
--! USE lemma subtract_duration_equals IN TRAIT
time_representation_conversions

    WITH
    temp,
      pair(
        (
          Int_identity(
            (the_interval.elapsed_days*days_to_seconds))
        +
          Int_identity(
            (the_interval.elapsed_hours*hours_to_seconds))),
        1),
      pair((the_interval.elapsed_hours*seconds_per_hour), 1)
    FOR
    m:AnyRecordSort, n:AnyRecordSort, p:AnyRecordSort
--! substituting for left;
--! USE lemma good_duration_pair IN TRAIT
time_representation_conversions

    WITH
    (the_interval.elapsed_hours*seconds_per_hour), 1
    FOR
    n:Int, d:Int;
--! USE lemma good_duration_sum IN TRAIT
time_representation_conversions

    WITH
    temp,
      pair((the_interval.elapsed_hours*seconds_per_hour), 1)
    FOR
    d1:AnyRecordSort, d2:AnyRecordSort;
--! USE lemma seconds_to_duration IN TRAIT sort_names

    WITH
    (the_interval.elapsed_hours*seconds_per_hour)
    FOR
    the_second:Int
--! substituting for left;
temp:=(temp+
  builtin.duration_ize((
    integer_ize(the_interval.elapsed_hours)*
```

```
    seconds_per_hour)));
--! VC Status: proved
--! BY lemma pair_minus in trait fixed_point
--!      WITH
  (
    (
      Int_identity(
        (the_interval.elapsed_days*days_to_seconds))
    +
      Int_identity(
        (the_interval.elapsed_hours*hours_to_seconds)))
  +
    Int_identity(
      (the_interval.elapsed_minutes*minutes_to_seconds))),
  1, (the_interval.elapsed_minutes*seconds_per_minute), 1
  FOR n1:Int, d1:Int, n2:Int, d2:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH
  (
    (
      Int_identity(
        (the_interval.elapsed_days*days_to_seconds))
    +
      Int_identity(
        (the_interval.elapsed_hours*hours_to_seconds)))
  +
    Int_identity(
      (the_interval.elapsed_minutes*minutes_to_seconds)))
  FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH
  (the_interval.elapsed_minutes*seconds_per_minute) FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH
  (the_interval.elapsed_minutes*minutes_to_seconds) FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_hours*hours_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
```

```
--!      WITH (the_interval.elapsed_days*days_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY contradiction, in 1
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_days*days_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_hours*hours_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY simplification
--! BY synthesis of TRUE
--! □
--| ASSERT (
  (
    (
      temp
    =
      pair(
        (
          Int_identity(
            (the_interval.elapsed_days*days_to_seconds))
        +
          Int_identity(
            (the_interval.elapsed_hours*hours_to_seconds))),
        1))
   AND
    good_interval(the_interval))
 AND
  (
    (
      good_duration(temp)
     AND
      (seconds_per_minute=minutes_to_seconds))
   AND
    (
      good_interval(the_interval)
     AND
      good_second(
        (the_interval.elapsed_milliseconds/1000)))));
--! USE lemma subtract_duration_equals IN TRAIT
time_representation_conversions
```

```
WITH
temp,
  pair(
    (
      (
        Int_identity(
          (the_interval.elapsed_days*days_to_seconds))
      +
        Int_identity(
          (the_interval.elapsed_hours*hours_to_seconds)))
    +
      Int_identity(
        (
          the_interval.elapsed_minutes
        *
          minutes_to_seconds))), 1),
  pair(
    (the_interval.elapsed_minutes*seconds_per_minute),
    1)
FOR
m:AnyRecordSort, n:AnyRecordSort, p:AnyRecordSort
--! substituting for left;
--! PRECONDITION = (good_duration(temp)
AND
(good_duration(temp)
  AND
  ((((temp
        +

          pair(((the_interval.elapsed_minutes)
              *
              seconds_per_minute),
            1))
        =

        pair(((
              Int_identity(((the_interval.elapsed_days)
                  *
                  days_to_seconds))
              +

              Int_identity(((the_interval.elapsed_hours)
                  *
```

```
                    hours_to_seconds)))
                +

                Int_identity(((the_interval.elapsed_minutes)
                    *
                    minutes_to_seconds))),
              1))
          AND
          good_interval(the_interval))
        AND
        (good_interval(the_interval)
          AND

          good_second(((the_interval.elapsed_milliseconds)
              /
              1000))))));
--! USE lemma good_duration_pair IN TRAIT
time_representation_conversions

    WITH
    (the_interval.elapsed_minutes*seconds_per_minute), 1
    FOR
    n:Int, d:Int;
--! USE lemma good_duration_sum IN TRAIT
time_representation_conversions

    WITH
    temp,
      pair(
        (the_interval.elapsed_minutes*seconds_per_minute),
        1)
    FOR
    d1:AnyRecordSort, d2:AnyRecordSort;
--! USE lemma seconds_to_duration IN TRAIT sort_names

    WITH
    (the_interval.elapsed_minutes*seconds_per_minute)
    FOR
    the_second:Int
--! substituting for left;
temp:=(temp+
  builtin.duration_ize((
    integer_ize(the_interval.elapsed_minutes)*
    seconds_per_minute)));
```

```
--! VC Status: proved
--! BY lemma pair_minus in trait fixed_point
--!      WITH
  (
    (
      (
        Int_identity(
          (the_interval.elapsed_days*days_to_seconds))
      +
        Int_identity(
          (the_interval.elapsed_hours*hours_to_seconds)))
    +
      Int_identity(
        (the_interval.elapsed_minutes*minutes_to_seconds)))
  +
    Int_identity(the_interval.elapsed_seconds)), 1,
  the_interval.elapsed_seconds, 1 FOR n1:Int, d1:Int, n2:
  Int, d2:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH
  (
    (
      (
        Int_identity(
          (the_interval.elapsed_days*days_to_seconds))
      +
        Int_identity(
          (the_interval.elapsed_hours*hours_to_seconds)))
    +
      Int_identity(
        (the_interval.elapsed_minutes*minutes_to_seconds)))
  +
    Int_identity(the_interval.elapsed_seconds)) FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_days*days_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_hours*hours_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
```

```
--!      WITH
  (the_interval.elapsed_minutes*minutes_to_seconds) FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH the_interval.elapsed_seconds FOR n:Int
--!      substituting for left
--! BY contradiction, in 1
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_days*days_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH (the_interval.elapsed_hours*hours_to_seconds)
  FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH
  (the_interval.elapsed_minutes*minutes_to_seconds) FOR n:Int
--!      substituting for left
--! BY simplification
--! BY synthesis of TRUE
--! □
--| ASSERT (
  (
    (
      temp
    =
      pair(
        (
          (
            Int_identity(
              (the_interval.elapsed_days*days_to_seconds))
          +
            Int_identity(
              (the_interval.elapsed_hours*hours_to_seconds)))
        +
          Int_identity(
            (
              the_interval.elapsed_minutes
            *
              minutes_to_seconds))), 1))
   AND
    (good_interval(the_interval) AND good_duration(temp)))
 AND
```

```
    (
      good_interval(the_interval)
     AND
      good_second((the_interval.elapsed_milliseconds/1000))));
  --! USE lemma subtract_duration_equals IN TRAIT
  time_representation_conversions

      WITH
      temp,
        pair(
          (
            (
              (
                Int_identity(
                  (the_interval.elapsed_days*days_to_seconds))
              +
                Int_identity(
                  (
                    the_interval.elapsed_hours
                  *
                    hours_to_seconds)))
            +
              Int_identity(
                (
                  the_interval.elapsed_minutes
                *
                  minutes_to_seconds)))
          +
            Int_identity(the_interval.elapsed_seconds)), 1),
        pair(the_interval.elapsed_seconds, 1)
      FOR
      m:AnyRecordSort, n:AnyRecordSort, p:AnyRecordSort
  --! substituting for left;
  --! USE lemma good_duration_pair IN TRAIT
  time_representation_conversions
   WITH the_interval.elapsed_seconds, 1 FOR n:Int, d:Int;
  <statement>
  --! USE lemma good_duration_sum IN TRAIT
  time_representation_conversions

      WITH
      temp, pair(the_interval.elapsed_seconds, 1)
      FOR
      d1:AnyRecordSort, d2:AnyRecordSort;
```

```
--! USE lemma seconds_to_duration IN TRAIT sort_names
 WITH the_interval.elapsed_seconds FOR the_second:Int
--! substituting for left;
temp:=(temp+
  builtin.duration_ize(the_interval.elapsed_seconds));
--! VC Status: ** not proved **
--! BY left substitution of 1
--! BY axiom Int_identity in trait Z
--!      WITH
  (
    (
      (
        Int_identity(
          (the_interval.elapsed_days*days_to_seconds))
      +
        Int_identity(
          (the_interval.elapsed_hours*hours_to_seconds)))
    +
      Int_identity(
        (the_interval.elapsed_minutes*minutes_to_seconds)))
  +
    Int_identity(the_interval.elapsed_seconds)) FOR n:Int
--!      substituting for left
--! BY axiom Int_identity in trait Z
--!      WITH
  (
    (
      (
        (
          (
            Int_identity(
              (the_interval.elapsed_days*days_to_seconds))
          +
            Int_identity(
              (the_interval.elapsed_hours*hours_to_seconds)))
        +
          Int_identity(
            (
              the_interval.elapsed_minutes
            *
              minutes_to_seconds)))
      +
        Int_identity(the_interval.elapsed_seconds))
    *
```

```
      1000)
   +
     Int_identity(the_interval.elapsed_milliseconds)) FOR n:
   Int
--!      substituting for left
--! BY claiming (
   Int_identity(
     ((the_interval.elapsed_milliseconds/1000)*1000))
=
   Int_identity(the_interval.elapsed_milliseconds))
--!    BY axiom Int_identity in trait Z
--!         WITH
   ((the_interval.elapsed_milliseconds/1000)*1000) FOR n:Int
--!         substituting for left
--!    BY axiom Int_identity in trait Z
--!         WITH the_interval.elapsed_milliseconds FOR n:Int
--!         substituting for left
--!    1. (temp
   =

   pair((((
          Int_identity(((the_interval.elapsed_days)
              *
              days_to_seconds))
          +

          Int_identity(((the_interval.elapsed_hours)
              *
              hours_to_seconds)))
        +

        Int_identity(((the_interval.elapsed_minutes)
            *
            minutes_to_seconds)))
      +
      Int_identity((the_interval.elapsed_seconds))),
    1))
--!    2. good_interval(the_interval)
--!    3. good_duration(temp)
--!    4.
good_second(((the_interval.elapsed_milliseconds)/1000))
--!    >> ((((the_interval.elapsed_milliseconds)/1000)*1000)
  =
  (the_interval.elapsed_milliseconds))
```

```
--!    <proof>
--! THEN
--!    BY claiming (
  pair(
    (
      (
        (
          Int_identity(
            (the_interval.elapsed_days*days_to_seconds))
        +
          Int_identity(
            (the_interval.elapsed_hours*hours_to_seconds)))
      +
        Int_identity(
          (the_interval.elapsed_minutes*minutes_to_seconds)))
    +
      Int_identity(the_interval.elapsed_seconds)), 1)
=
  pair(
    (
      (
        (
          (
            Int_identity(
              (the_interval.elapsed_days*days_to_seconds))
          +
            Int_identity(
              (the_interval.elapsed_hours*hours_to_seconds)))
        +
          Int_identity(
            (
              the_interval.elapsed_minutes
            *
              minutes_to_seconds)))
      +
        Int_identity(the_interval.elapsed_seconds))
    *
      1000), 1000))
--!       BY lemma concrete_equality_pair
                in trait fixed_point
--!             WITH
  (
    (
      (
```

```
          Int_identity(
            (the_interval.elapsed_days*days_to_seconds))
        +
          Int_identity(
            (the_interval.elapsed_hours*hours_to_seconds)))
      +
        Int_identity(
          (the_interval.elapsed_minutes*minutes_to_seconds)))
    +
      Int_identity(the_interval.elapsed_seconds)), 1,
    (
      (
        (
          (
            Int_identity(
              (the_interval.elapsed_days*days_to_seconds))
          +
            Int_identity(
              (the_interval.elapsed_hours*hours_to_seconds)))
        +
          Int_identity(
            (the_interval.elapsed_minutes*minutes_to_seconds)))
      +
        Int_identity(the_interval.elapsed_seconds))
    *
      1000), 1000 FOR n1:Int, d1:Int, u2:Int, d2:Int
--!             substituting for left
--!         BY synthesis of TRUE
--!      THEN
--!         BY left substitution of 5
--!         BY simplification
--!         BY synthesis of TRUE
--! □
--| ASSERT (
    (
      (
        (
          temp
        =
          pair(
            (
              (
                (
                  Int_identity(
```

```
                    (
                      the_interval.elapsed_days
                    *
                      days_to_seconds))
                  +
                    Int_identity(
                      (
                        the_interval.elapsed_hours
                      *
                        hours_to_seconds)))
                +
                  Int_identity(
                    (
                      the_interval.elapsed_minutes
                    *
                      minutes_to_seconds)))
              +
                Int_identity(the_interval.elapsed_seconds)), 1))
       AND
        good_interval(the_interval))
     AND
      good_duration(temp))
   AND
    good_second((the_interval.elapsed_milliseconds/1000)));
--! USE lemma pair_minus IN TRAIT fixed_point

      WITH

        (
          (
            (
              (
                (
                  Int_identity(
                    (
                      the_interval.elapsed_days
                    *
                      days_to_seconds))
                  +
                    Int_identity(
                      (
                        the_interval.elapsed_hours
                      *
                        hours_to_seconds)))
```

```
            +
              Int_identity(
                (
                  the_interval.elapsed_minutes
                *
                  minutes_to_seconds)))
          +
            Int_identity(the_interval.elapsed_seconds))
        *
          1000)
      +
        Int_identity(the_interval.elapsed_milliseconds)),
      1000, (the_interval.elapsed_milliseconds/1000), 1
    FOR
    n1:Int, d1:Int, n2:Int, d2:Int
--! substituting for left;
--! USE lemma subtract_duration_equals IN TRAIT
time_representation_conversions

    WITH
    temp,
      pair(
        (
          (
            (
              (
                (
                  Int_identity(
                    (
                      the_interval.elapsed_days
                    *
                      days_to_seconds))
                  +
                    Int_identity(
                      (
                        the_interval.elapsed_hours
                      *
                        hours_to_seconds)))
                +
                  Int_identity(
                    (
                      the_interval.elapsed_minutes
                    *
                      minutes_to_seconds)))
```

```
            +
              Int_identity(the_interval.elapsed_seconds))
          *
            1000)
        +
          Int_identity(the_interval.elapsed_milliseconds)),
        1000),
      pair((the_interval.elapsed_milliseconds/1000), 1)
    FOR
    m:AnyRecordSort, n:AnyRecordSort, p:AnyRecordSort
--! substituting for left;
--! USE lemma seconds_to_duration IN TRAIT sort_names

    WITH
    (the_interval.elapsed_milliseconds/1000)
    FOR
    the_second:Int
--! substituting for left;
--! USE axiom good_seconds_to_good_duration IN TRAIT
time_representation_conversions

    WITH
    (the_interval.elapsed_milliseconds/1000)
    FOR
    the_second:Int;
--! USE lemma good_duration_sum IN TRAIT
time_representation_conversions

    WITH
    temp,
      seconds_to_duration(
        (the_interval.elapsed_milliseconds/1000))
    FOR
    d1:AnyRecordSort, d2:AnyRecordSort;
--! USE axiom Int_identity IN TRAIT Z

    WITH

      (
        (
          (
            (
              Int_identity(
                (the_interval.elapsed_days*days_to_seconds))
```

```
            +
              Int_identity(
                (
                  the_interval.elapsed_hours
                *
                  hours_to_seconds)))
          +
            Int_identity(
              (
                the_interval.elapsed_minutes
              *
                minutes_to_seconds)))
        +
          Int_identity(the_interval.elapsed_seconds))
      *
        1000) FOR n:Int
--! substituting for left;
--! PRECONDITION = (
    good_duration(
      seconds_to_duration(((the_interval.elapsed_milliseconds
            )
          /
          1000)))
    ->
    ((good_duration(temp)
        AND

        good_duration(
          seconds_to_duration(
            (
              (the_interval.elapsed_milliseconds)
              /
              1000))))
     AND
     (
       good_duration((temp
           +

           seconds_to_duration(
             (
               (the_interval.elapsed_milliseconds)
             /
               1000))))
      ->
```

```
        (good_interval(the_interval)
          AND
          ((temp
              +

              seconds_to_duration(
                (
                  (the_interval.elapsed_milliseconds)
                /
                  1000)))
            =

            pair((
                Int_identity(((((
                          Int_identity(
                            ((the_interval.elapsed_days)
                              *
                              days_to_seconds))
                          +

                          Int_identity(
                            ((the_interval.elapsed_hours)
                              *
                              hours_to_seconds)))
                        +

                        Int_identity(
                          ((the_interval.elapsed_minutes)
                            *
                            minutes_to_seconds)))
                      +

                      Int_identity(
                        (the_interval.elapsed_seconds)))
                    *
                    1000))
                +

                Int_identity(
                  (the_interval.elapsed_milliseconds))),
              1000))))));
temp:=(temp+
  builtin.duration_ize((the_interval.elapsed_milliseconds/
    1000)));
```

```
--! USE lemma pair_plus IN TRAIT fixed_point

    WITH

      (
        (
          (
            Int_identity(
              (the_interval.elapsed_days*days_to_seconds))
          +
            Int_identity(
              (the_interval.elapsed_hours*hours_to_seconds)))
        +
          Int_identity(
            (
              the_interval.elapsed_minutes
            *
              minutes_to_seconds)))
      +
        Int_identity(the_interval.elapsed_seconds)), 1,
      the_interval.elapsed_milliseconds, 1000
    FOR
    n1:Int, d1:Int, n2:Int, d2:Int
--! substituting for left;
--! USE axiom Int_identity IN TRAIT Z

    WITH

      (
        (
          Int_identity(
            (the_interval.elapsed_days*days_to_seconds))
        +
          Int_identity(
            (the_interval.elapsed_hours*hours_to_seconds)))
      +
        Int_identity(
          (
            the_interval.elapsed_minutes
          *
            minutes_to_seconds)))
    FOR
    n:Int
--! substituting for left;
```

```
--! USE lemma pair_plus IN TRAIT fixed_point

    WITH

      (
        (
          Int_identity(
            (the_interval.elapsed_days*days_to_seconds))
        +
          Int_identity(
            (the_interval.elapsed_hours*hours_to_seconds)))
      +
        Int_identity(
          (
            the_interval.elapsed_minutes
          *
            minutes_to_seconds))), 1,
      the_interval.elapsed_seconds, 1
    FOR
    n1:Int, d1:Int, n2:Int, d2:Int
--! substituting for left;
--! USE axiom Int_identity IN TRAIT Z

    WITH

      (
        Int_identity(
          (the_interval.elapsed_days*days_to_seconds))
      +
        Int_identity(
          (the_interval.elapsed_hours*hours_to_seconds)))
    FOR
    n:Int
--! substituting for left;
--! USE lemma pair_plus IN TRAIT fixed_point

    WITH

      (
        Int_identity(
          (the_interval.elapsed_days*days_to_seconds))
      +
        Int_identity(
          (the_interval.elapsed_hours*hours_to_seconds))),
```

```
      1, (the_interval.elapsed_minutes*minutes_to_seconds),
      1
    FOR
    n1:Int, d1:Int, n2:Int, d2:Int
--! substituting for left;
--! USE lemma pair_plus IN TRAIT fixed_point

    WITH
    (the_interval.elapsed_days*days_to_seconds), 1,
      (the_interval.elapsed_hours*hours_to_seconds), 1
    FOR
    n1:Int, d1:Int, n2:Int, d2:Int
--! substituting for left;
--! USE axiom seconds_to_milliseconds IN TRAIT
conversion_factors
--! substituting for left;
--! USE lemma divides_pi IN TRAIT fixed_point

    WITH
    the_interval.elapsed_milliseconds, 1,
      seconds_to_milliseconds FOR n:Int, d:Int, i:Int
--! substituting for left;
--! USE lemma seconds_to_duration IN TRAIT sort_names

    WITH
    (the_interval.elapsed_days*days_to_seconds)
    FOR
    the_second:Int
--! substituting for left;
--! USE lemma seconds_to_duration IN TRAIT sort_names

    WITH
    (the_interval.elapsed_hours*hours_to_seconds)
    FOR
    the_second:Int
--! substituting for left;
--! USE lemma seconds_to_duration IN TRAIT sort_names

    WITH
    (the_interval.elapsed_minutes*minutes_to_seconds)
    FOR
    the_second:Int
--! substituting for left;
--! USE lemma seconds_to_duration IN TRAIT sort_names
```

```
   WITH the_interval.elapsed_seconds FOR the_second:Int
  --! substituting for left;
  --! USE lemma seconds_to_duration IN TRAIT sort_names
   WITH the_interval.elapsed_milliseconds FOR the_second:Int
  --! substituting for left;
  --! USE axiom interval_to_duration IN TRAIT
  time_representation_conversions
   WITH the_interval FOR the_interval:AnyRecordSort
  --! substituting for left;
  --! SIMPLIFIED PRECONDITION;
  --! USE axiom good_interval_to_good_duration IN TRAIT
  time_representation_conversions

      WITH
      the_interval, temp
      FOR
      the_interval:AnyRecordSort, the_duration:AnyRecordSort;
  --! PRECONDITION = ((temp
        =
        interval_to_duration(the_interval))
      AND
      good_duration(temp));
  RETURN  temp;
END duration_of;
FUNCTION interval_of(the_duration : IN builtin.duration) RETURN
 interval
  --| WHERE
  --|      GLOBAL seconds_per_day, seconds_per_hour,
  seconds_per_minute : IN ;
  --|      IN good_duration(the_duration);
  --|      RETURN duration_to_interval(the_duration);
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  result : interval;
  the_seconds : builtin.duration := the_duration;

BEGIN
  result.elapsed_days:=duration_utilities.floor((the_seconds/
    seconds_per_day));
  the_seconds:=(the_seconds-
    builtin.duration_ize((integer_ize(result.elapsed_days)*
```

```
        seconds_per_day)));
    result.elapsed_hours:=hour_ize(duration_utilities.floor((
      the_seconds/seconds_per_hour)));
    the_seconds:=(the_seconds-
      builtin.duration_ize((integer_ize(result.elapsed_hours)*
        seconds_per_hour)));
    result.elapsed_minutes:=minute_ize(duration_utilities.floor((
      the_seconds/seconds_per_minute)));
    the_seconds:=(the_seconds-
      builtin.duration_ize((integer_ize(result.elapsed_minutes)*
        seconds_per_minute)));
    result.elapsed_seconds:=
      second_ize(duration_utilities.floor(the_seconds));
    the_seconds:=(the_seconds-
      builtin.duration_ize(result.elapsed_seconds));
    --! PRECONDITION = (((Int_to_fixnum(0)<=(the_seconds*1000))
          ->
          (0<=Floor((the_seconds*1000))))
        ->
        (good_duration(the_seconds)
          AND
          (good_duration((the_seconds*1000))
            ->
            (((0<=Floor((the_seconds*1000)))
                AND
                (Floor((the_seconds*1000))<1000))
              AND
              ((result[.elapsed_milliseconds=>
                  Floor((the_seconds*1000))])
                =
                duration_to_interval(the_duration))))));
    --! INSTANTIATE lemma floor1 IN TRAIT fixed_point
     WITH (the_seconds*1000) FOR m:AnyRecordSort;
    result.elapsed_milliseconds:=
      millisecond_ize(
        duration_utilities.floor((the_seconds*1000)));
    RETURN  result;
  END interval_of;
END calendar_utilities;
```

References

[1] Dan Craigen et al. m-EVES: A tool for verifying software.. In *Proceedings of the 10th International Conference on Software Engineering, Singapore*, April 1988.

[2] J.S. Crow et al. EHDM verification environment: An overview. In *Proceedings of the 11th National Computer Security Conference*, 1988.

[3] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, 1976.

[4] FAA. System Design and Analysis. Advisory Circular AC 25.1309-1A, U.S. Department of Transportation, June 1988.

[5] Donald I. Good, Robert L. Akers, and Lawrence M. Smith. Report on Gypsy 2.05. Technical report, Computational Logic Inc., 1986.

[6] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[7] David Guaspari. Domains for Ada types. Technical report, ORA Corporation, September 1989.

[8] David Guaspari. Penelope, an Ada verification system. In *Proceedings of Tri-Ada '89*, pages 216–224, Pittsburgh, PA, October 1989.

[9] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16:1058–1075, September 1990.

[10] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report TR 5, DEC/SRC, July 1985.

[11] Bret Hartman, Douglas Hoover, and Sanjiva Prasad. Final report on specification language. Technical Report BOA 3695.STARS-043, ORA Corporation, February 1991.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[13] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[14] Carla Marceau and C. Douglas Harper. An interactive approach to Ada verification. In *Proceedings of the 12th National Computer Security Conference*, pages 28–51, Baltimore, MD, October 1989.

[15] L.G. Marcus and J.V. Cook. SDVS user's manual. Technical Report ATR-84(8478)-1, Aerospace Corporation, El Segundo, Calif. 90245, November 1984.

[16] Norman Ramsey. Developing formally verified Ada programs. In *Proceedings of the Fifth International Conference on Software Specification and Design*, May 1989.

[17] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1993	Contractor Report

4. TITLE AND SUBTITLE Using Penelope to Assess the Correctness of NASA Ada Software: A Demonstration of Formal Methods as a Counterpart to Testing

5. FUNDING NUMBERS
C NAS1-18972
WU 505-64-10-05

6. AUTHOR(S) Carl T. Eichenlaub, C. Douglas Harper, and Geoffrey Hird

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ORA Corporation
301A Dates Drive
Ithaca, NY 14850-1313

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR-4509

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Ricky W. Butler
Final Report

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited

Star Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

Life-critical applications warrant a higher level of software reliability than has yet been achieved. Since it is not certain that traditional methods alone can provide the required ultrareliability, new methods should be examined as supplements or replacements. This paper describes a mathematical counterpart to the traditional process of empirical testing.

ORA's Penelope verification system is demonstrated as a tool for evaluating the correctness of Ada software. Grady Booch's Ada calendar utility package, obtained through NASA, was specified in the Larch/Ada language. Formal verification in the Penelope environment established that many of the package's subprograms met their specifications. In other subprograms, failed attempts at verification revealed several errors that had escaped detection by testing.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Life-critical software; Ultrareliability; Formal methods; Formal specification; Formal verification safety	156
	16. PRICE CODE A08

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500 • U.S GOVERNMENT PRINTING OFFICE 1993 .728 .064/66037

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102